МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ ІМЕНІ СЕМЕНА КУЗНЕЦЯ

А. О. Поляков
В. М. Федорченко
О. В. Шматко

АНАЛІЗ МЕТОДІВ І ТЕХНОЛОГІЙ РОЗРОБЛЕННЯ МОБІЛЬНИХ ДОДАТКІВ ДЛЯ ПЛАТФОРМИ ANDROID

Навчальний посібник

Харків
ХНЕУ ім. С. Кузнеця
2017

УДК 004.4(075.034)
П54

Авторський колектив: канд. техн. наук, доцент А. О. Поляков – підрозділи 1.4, 1.5, 2.6 – 2.8, розділ 3, підрозділи 4.3, 4.5; канд. техн. наук, доцент В. М. Федорченко – вступ, підрозділи 1.1 – 1.3, 2.4, 2.5, 4.2, 4.4; канд. техн. наук, доцент О. В. Шматко – підрозділи 2.1 – 2.3, 4.1.

Рецензенти: канд. техн. наук, доцент кафедри Програмної інженерії та інформаційних технологій управління Національного технічного університету "Харківський політехнічний інститут" *О. Ю. Чередніченко*; канд. техн. наук, доцент кафедри "Комп'ютерні системи і мережі" Національного аерокосмічного університету "ХАІ" *А. В. Шостак*.

Рекомендовано до видання рішенням ученої ради Харківського національного економічного університету імені Семена Кузнеця.
Протокол № 10 від 05.07.2017 р.

Самостійне електронне текстове мережеве видання

Поляков А. О.

П54 Аналіз методів і технологій розроблення мобільних додатків для платформи Android : навчальний посібник [Електронний ресурс] / А. О. Поляков, В. М. Федорченко, О. В. Шматко. – Харків : ХНЕУ ім. С. Кузнеця, 2017. – 286 с.

ISBN 978-966-676-698-7

Розкрито основи програмування для платформи Android. Розглянуто архітектуру платформи і ОС Android. Досліджено архітектуру мобільного програмного додатка. Розкрито сутність та призначення основних програмних компонентів: Activity, Service, ContentProvider і BroadcastReceiver. Наведено методи побудови інтуїтивного інтерфейсу користувача, а також достатню кількість пояснювальних прикладів. Розглянуто різні форми збереження та оброблення даних на мобільному пристрої.

Рекомендовано для студентів закладів вищої освіти.

УДК 004.4(075.034)

© А. О. Поляков, В. М. Федорченко, О. В. Шматко, 2017
© Харківський національний економічний університет імені Семена Кузнеця, 2017

ISBN 978-966-676-698-7

Вступ

Популярність використання мобільних пристроїв у всьому світі продовжує зростати. Сьогодні користувачі витрачають більше часу на свої смартфони в різних цілях (соціальні мережі, електронна пошта, карти, новини, відео, комерційні додатки та ін.). У таких умовах господарювання вимагає від фахівців з економічного управління всебічного використання новітніх інформаційних технологій. Широкі можливості мобільних засобів в питаннях збирання, оброблення та видачі необхідної інформації здатні значно підвищити якість економічних розрахунків, зробити більш ефективним процес обґрунтування економічних рішень.

Таким чином процес розроблення мобільних додатків стає актуальним напрямом у IT індустрії. Сучасні компанії, такі, як: Google, Apple, Microsoft та інші розробили мобільні платформи, що включають мобільні ОС та засоби розроблення (SDK, Software Developer Kit). Важливою особливістю мобільних пристроїв є те, що вони мають обмежене джерело живлення, невеликій розмір екрану та набір різноманітних сенсорів. Розроблення мобільних додатків достатньо технологічний процес, що потребує певних компетенцій з ООП, знання SQL, проектування БД та UI, розуміння мережної взаємодії, тестування ПЗ.

Найбільш розповсюдженою мобільною платформою є Android і складає понад 82 % від усіх засобів на 2016 р., що підтверджує її універсальність і перспективність вивчення даної платформи. Після вивчення матеріалу посібника студенти оволодіють такими компетенціями: проектувати та розробляти вимоги до мобільного додатка, з урахуванням особливостей мобільної платформи; визначати та класифікувати вимоги до розроблення мобільних додатків; проектувати мобільний інтерфейс користувача та розробляти мобільний додаток з урахуванням життєвого циклу компонентів; розробляти ефективні мобільні рішення під сучасні мобільні платформи; здійснювати обґрунтування прийняття проектних рішень (розробляти інтерактивні елементи управління мобільного додатка); проектувати та розробляти мобільний додаток з локальною обробкою даних та зовнішніми сховищами даних.

У посібнику розглядаються чотири теми, які слід вивчати поступово.

У першому розділі подані основні положення мобільної платформи, розглянуті процеси та інструменти розроблення для платформи Android,

її архітектура. Надається огляд віртуальних машин: Dalvik і ART, та розглядаються їх особливості і продуктивність.

У другому розділі надається огляд архітектурних моделей для розроблення мобільних прикладних програм, засобів їх розроблення та застосування інтегрованих середовищ розроблення (IDE). Пояснюються важливі поняття завдання та стеку переходів назад, запуск завдання. Досліджуються базові компоненти для створення інтерфейсу користувача і базові макети. Досліджується життєвий цикл візуальних компонентів. Розглядаються об'єкти та методи, що дозволяють виконувати комунікації між компонентами Android, а також компонентами інших додатків. Даються практичні рекомендації з використання випливних повідомлень.

У третьому розділі досліджується робота зі службами, що дозволяють використовувати у додатку тривалі процеси оброблення даних, що виконуються у фонових потоках та процесах. Досліджуються життєвий цикл служби, прив'язка служби, спеціалізована мова AIDL. Наведені практичні приклади створення служб.

Четвертий розділ присвячений вивченню компонентів оброблення та збереження даних як у локальному пристрої, так і в Cloud Storage. Наводяться основні відомості про постачальника контенту та розглядається процес створення постачальника контенту. Досліджується робота із завантажувачами (Loaders) та компонентом контент-провайдера. Розглядається платформа доступу до сховища (Storage Access Framework).

Розділ 1. Архітектура та компоненти мобільних платформ

Мета: надати основні принципи щодо архітектури та компонентів мобільних платформ.

Компетентності: Визначати та класифікувати вимоги до розроблення мобільних додатків. Здатність налагоджувати процес розроблення мобільного додатка. Визначати технології розроблення мобільного додатка.

Знати: архітектуру та компоненти мобільної платформи, процес взаємодії компонентів.

Вміти: розгортати та налагоджувати середовища розроблення для мобільних платформ Android і Microsoft Windows 10; створювати та налагоджувати емулятори мобільного пристрою; ефективно формувати комунікаційну стратегію під час роботи в команді.

Відповідати за якість налагодження середовища розроблення та строків його реалізації.

Основні питання:

1. Огляд операційного середовища Android.
2. Огляд структури Android-прикладної програми.
3. Віртуальна машина в платформі Android.
4. Управління ресурсами мобільних пристроїв.
5. API мобільних прикладних програм.

Ключові слова: платформа Android, уявлення, віртуальна машина, віджити, компіляція, JIT компіляція.

1.1. Огляд операційного середовища Android

Платформа Android є розробкою групи Open Handset Alliance, яка поставила собі за мету створити інноваційну модель телефону. Ця група на чолі з Google об'єднує операторів мобільних мереж, виробників телефонів і компонентів, розробників програмних рішень і постачальників послуг, а також маркетингові компанії. Таким чином, платформа Android є ядром розроблення програмного забезпечення з відкритим кодом.

Першим телефоном, який просувався на ринок оператором T-Mobile і працював на платформі Android був пристрій G1 від компанії "HTC". Цей телефон вийшов на ринок через рік після перших згадок про нього.

Для розроблення програмного забезпечення використовувався SDK, який постійно оновлювався. Напередодні випуску G1 команда Android анонсувала SDK v2.0, після чого почали з'являтися прикладні програми для нової платформи.

Щоб стимулювати інновації, Google спонсорувала два "Конкурси розробників для *Android*", переможці яких отримали мільйони доларів. Через кілька місяців після виходу G1 відкрився сайт Android Market, звідки користувачі могли завантажувати додатки прямо до свого телефону. Усього за півтора року нова мобільна платформа вийшла на арену.

1.2. Огляд структури Android-прикладної програми

За широтою можливостей платформа Android не поступається операційним системам настільних ПК. Це багаторівневе середовище на основі ядра Linux із великими функціональними можливостями. До підсистеми інтерфейсу, призначеного для користувача, входять:

- вікна;
- уявлення (відображення елементів інтерфейсу на екран);
- віджети для відображення таких загальних елементів, як поля, що можуть редагуватися, списки та розгортні списки.

В Android вбудовано браузер із движком WebKit із відкритим вихідним кодом, який лежить в основі браузера Safari мобільного телефону iPhone.

Android має широкий спектр можливостей для підключення, бездротового зв'язку з використанням Wi-Fi, Bluetooth та протоколів передачі даних через стільникову мережу (GPRS, EDGE, 3G і ін.). В Android активно використовують сервіси, які надає Google, наприклад, у своїх прикладних програмах можна посилатися на Google Maps для позиціонування пристрою. До стека програмного забезпечення Android входить також підтримка сервісів, заснованих на визначенні місця розташування (наприклад, GPS), та акселерометрів, хоча не всі пристрої на цій платформі оснащено необхідним обладнанням. Наявна також підтримка відеокамери.

Через обмеження ресурсів мобільні пристрої завжди мали великі проблеми з обробленням графіки/мультимедіа та способами зберігання даних, на відміну від персональних комп'ютерів. Для вирішення цієї проблеми платформа Android пропонує вбудовану підтримку 2D- і 3D- графіки з використанням бібліотеки OpenGL. Для забезпечення зберігання даних

платформа Android використовує популярну базу даних із відкритим вихідним кодом SQLite. На рис. 1.1 зображено спрощену схему рівнів програмного забезпечення Android.

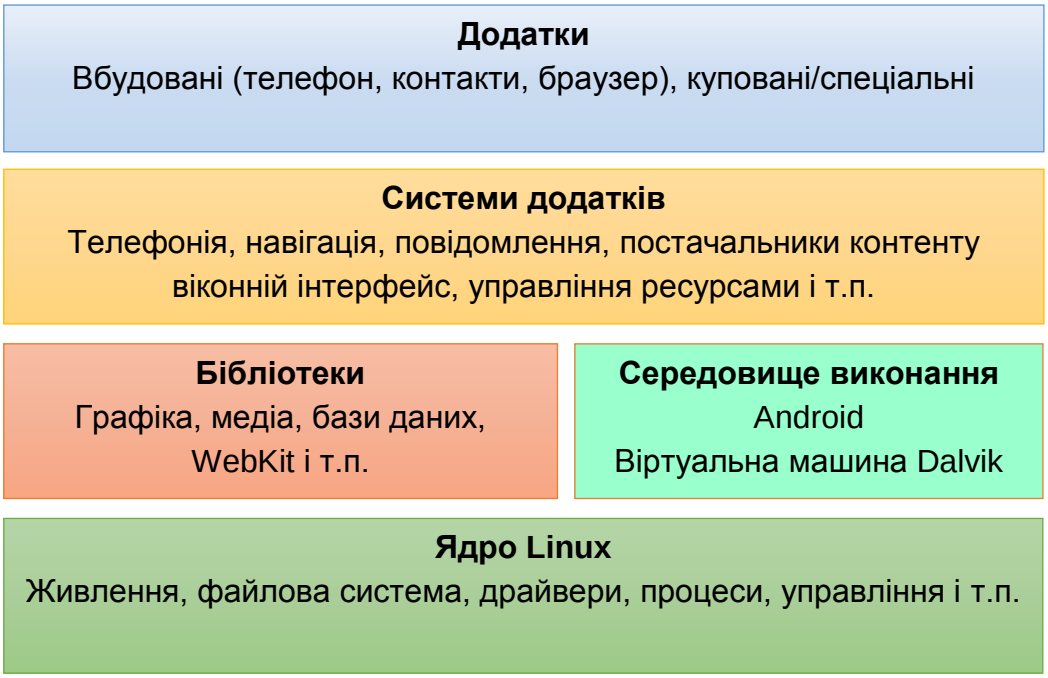

Рис. 1.1. **Рівні програмного забезпечення Android**

1.3. Огляд віртуальних машин Android

1.3.1. Віртуальна машина Dalvik

Операційна система Android працює зверх ядра Linux. Для створення Android-додатків спочатку використовували мову програмування Java, а прикладні програми виконували у віртуальній машині (VM). Необхідно звернути увагу на те, що віртуальна машина – це не віртуальна машина Java (JVM), а відкрита технологія Dalvik Virtual Machine. Під час запуску програми Android створюється та запускається окремий екземпляр Dalvik VM, який, у свою чергу, розташований у межах керованого ядром Linux процесу, як подано на рис. 1.2.

Dalvik є віртуальною машиною (VM) у межах операційної системи Android від Google, яка виконує прикладні програми, написані для Android. Dalvik є невід'ємною частиною стека програмного забезпечення Android в ОС Android версії 4.4 KitKat та більш ранніх версій, які зазвичай використовують у таких мобільних пристроях, як мобільні телефони та планшетні комп'ютери, а останнім часом на смарт-телевізорах. Dalvik є про-

грамним забезпеченням із відкритим вихідним кодом, написаним Dan Bornstein, який названий на честь рибальського селища Dalvík в Ісландії.

Програми для Android, переважно, написані на Java та скомпільовані в байт-код для віртуальної машини Java, яка потім транслюється на Dalvik байт-код і зберігається в .dex (Dalvik Executable) і .odex (оптимізоване Dalvik Executable) файлах. Компактний формат Dalvik Executable призначено для систем, обмежених із точки зору пам'яті та швидкості, процесора.

Правонаступник Dalvik є Android Runtime (ART), який використовує той же байт-код та .dex файли (але не .odex файли), із послідовністю, спрямованою на підвищення продуктивності для кінцевих користувачів.

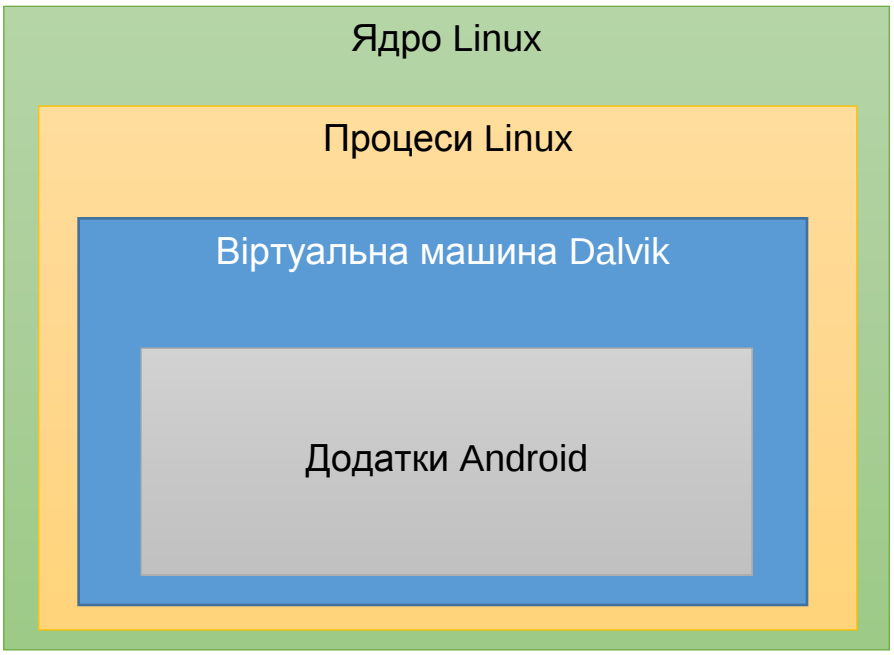

Рис. 1.2. **Dalvik VM**

1.3.2. Віртуальна машина ART

Android Runtime (ART) є середовищем виконання прикладних програм, яке використовує Android. ART виконує перетворення байт-коду програми в інструкції, які потім виконують за допомогою середовища виконання пристрою.

В Android 2.2 Froyo з'явилися JIT-компіляція в Dalvik, оптимізація виконання прикладних програм за допомогою постійно профілюючих програм. Тоді як Dalvik інтерпретує частину в байт-коді прикладної програми,

виконання ART цих коротких відрізків байт-коду, що називають "слідом", забезпечує значне підвищення продуктивності.

На відміну від Dalvik, ART вводить у використання ahead-of-time (AOT) компіляцію шляхом компіляції цілих прикладних програм у машинний код під час їхнього встановлення. Усуваючи інтерпретації Dalvik на основі JIT-компіляції, ART підвищує загальну ефективність виконання та знижує енергоспоживання, що приводить до підвищення автономії батареї на мобільних пристроях. Також ART забезпечує більш швидке виконання програм, поліпшений розподіл пам'яті та механізми збирання сміття (GC), нові можливості налагодження прикладних програм, а також більш точне профілювання прикладних програм високого рівня.

Для забезпечення зворотної сумісності ART використовує такий же вхідний байт-код, що і Dalvik, який надається через стандартні .dex файли як частина APK файлів, тоді як .odex файли замінюються Executable and Linkable Format (ELF) файлами (рис. 1.3). Після того як прикладну програму буде зібрано та скомпільовано на пристрої, утиліта dex2oat запускається з виконуваного ELF-файла. У результаті ART усуває відмінності виконання прикладних програм, накладні витрати, пов'язані з інтерпретацією Dalvik, та трасування на основі JIT-компіляції.

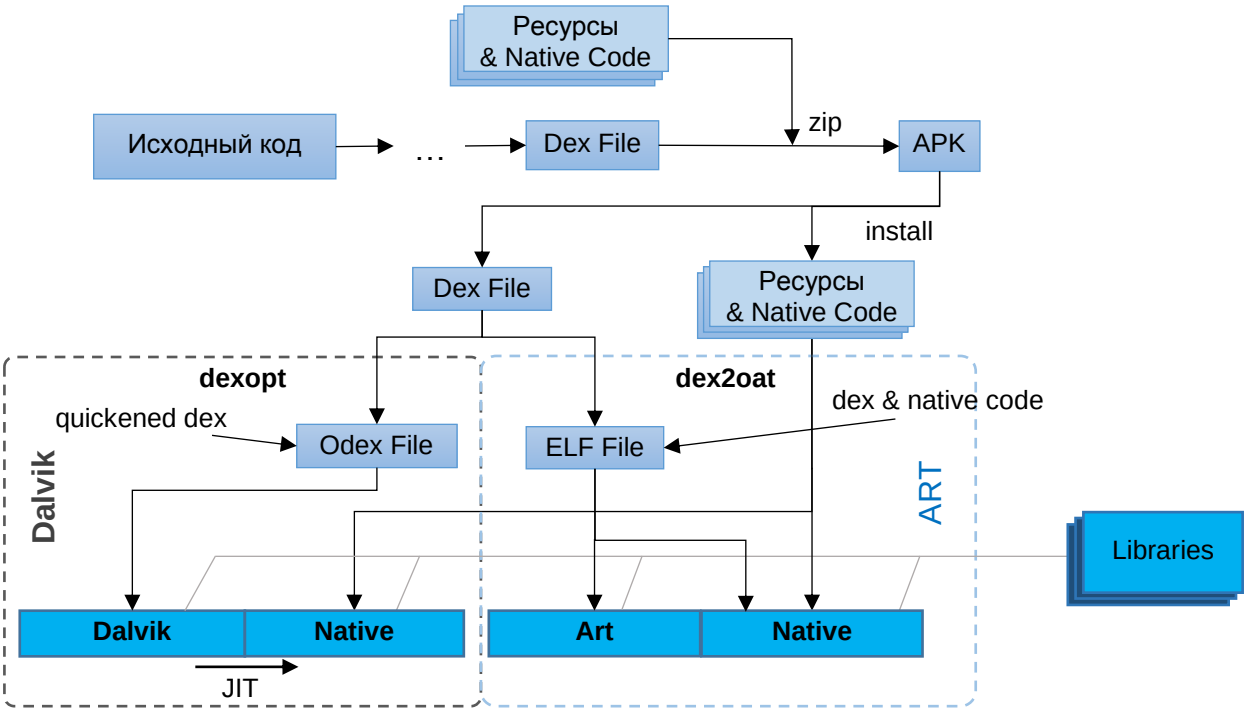

Рис. 1.3. **Порівняння архітектури Dalvik та ART**

ART потребує додаткового часу для компіляції під час установлення прикладної програми, а також прикладні програми – трохи більше місця для зберігання (переважно, використовують флеш-пам'ять) скомпільованого коду.

В Android 4.4 KitKat з'явилася технологія попереднього огляду ART, у тому числі як альтернативного середовища виконання та збереження замість Dalvik віртуальної машини поза вибором. У подальшому в Android 5.0 Lollipop, Dalvik було повністю замінено на ART.

Android-прикладна програма містить елементи одного або декількох таких типів, як:

1. Дії (Activities).

2. Прикладна програма із графічним інтерфейсом, що реалізується за допомогою дії. Коли користувач вибирає прикладну програму на головному екрані або екрані запуску програм, він викликає дію.

3. Сервіси (Services).

Сервіси використовують для таких програм, які працюють протягом тривалого часу, як мережевий монітор або перевірка оновлень.

4. Джерела даних (Content providers).

Джерело даних можна уявити собі як сервер баз даних. Його завдання – управління доступом до даних, що зберігають, наприклад баз даних SQLite. Якщо прикладна програма зовсім проста, джерело даних створювати не обов'язково. Якщо пишуть більш складну прикладну програму або прикладну програму, у якій до даних звернено кілька дій або прикладних програм, джерело даних є засобом організації доступу до вашої інформації.

5. Приймачі (Broadcast receivers).

Android-прикладну програму можна запускати для оброблення елемента даних або реагування на події, наприклад, на отримання текстового повідомлення.

Прикладна програма для Android розгортається на пристрої разом із файлом AndroidManifest.xml. Цей файл містить необхідну інформацію про конфігурацію, яка дозволяє правильно встановити прикладну програму на пристрої. Він містить також необхідні назви класів та типи подій, які може обробляти прикладна програма, та дозволи, необхідні для її роботи. Так, якщо із прикладною програмою потрібен доступ до мережі (наприклад, щоб завантажити файл) відповідний дозвіл має бути явно вказано у файлі-маніфесті. Цей конкретний дозвіл можуть мати багато

прикладних програм. Такий захист шляхом декларування допомагає зменшити ймовірність пошкодження пристрою з вини некоректно написаної прикладної програми.

1.4. Управління ресурсами мобільних пристроїв

Ядро, що використовують в Android, є модернізованим ядром серії Linux для забезпечення виконання деяких особливих потреб мобільних платформ. Функції ядра в основному поширено на драйвер, управління живленням, засоби управління та коригування обмежених можливостей Android. Функції управління живленням мають вирішальне значення для мобільних пристроїв.

Ядро Android перебуває у вільному доступі. Усі зміни можна відстежувати через загальнодоступне сховище Android. Є кілька ядер, доступних у репозиторії. Деякі апаратні специфічні ядра для таких платформ, як MSM7xxx, Open Multimedia Application Platform (OMAP) та Tegra, також доступні у сховищі.

Зміни, які вносять до базової версії ядра, можна розподілити на: виправлення помилок, засоби для підвищення простору для користувача (low memory killer, binder, ash mem, logger і т. д.), нову інфраструктуру (особливо wake locks), підтримку нових SoCs (msm7k, msm8k і т. д.) та плат/пристроїв. У майбутньому специфічні ядра Android та базові ядра Linux мають об'єднатися, але цей процес іде повільно та забере деякий час.

Android виділяє кожній прикладній програмі, призначеній для користувача, окремий адресний простір. Програми, що працюють у Dalvik VM або ART періодично переходять у сплячий режим або режим очікування. Це важливо, тому що поза вибором, Android намагається перевести систему в режим сну або режим очікування якнайшвидше.

При цьому екран залишається ввімкненим або CPU не спить, щоб швидше реагувати на пробудження. Інструменти Android, які використовують для вирішення цього завдання, називають блокіраторами засипання (wakelocks).

Функції `wakelock` можуть бути отримані компонентами ядра або адресним простором процесу користувача.

Інтерфейс користувача для створення блокіраторів використовує файл `/sys/power/wake_lock`, у якому записується назва нового блокіратора.

Для зняття блокування, процес записує назву файла в `/sys/power/wake_unlock`. Для блокування можна вказати час, за якого воно спрацює автоматично. Усі системи, які використовують на цей час сервіс блокування, указуються у файлі `/proc/wakelocks`.

Інтерфейс ядра для блокіраторів пробудження дозволяє вказати, чи має `Wake_lock` запобігати низькому енергоспоживанню або припиненню роботи системи. `wake_lock` створюється процесом `wake_lock_init()` та вилучається wake `lock_destroy()`. Створений блокіратор може бути запущений процесом `wake_lock()` та розблокований `wake_unlock()`. Як і у просторі користувача можна визначити тайм-аут для блокування. Концепцію блокіраторів глибоко інтегровано в Android у вигляді драйверів, багато прикладних програм інтенсивно використовують їх.

Клас `PowerManager` – це службовий клас, який дозволяє віртуальним машинам Андроїд отримати доступ до можливостей WakeLock драйвера управління живленням. `PowerManager` забезпечує чотири різних види блокування:

- `PARTIAL WAKE LOCK` – CPU не спить, навіть якщо кнопку живлення пристрою натиснуто.
- `SCREEN DIM WAKE LOCK` – блокування екрану залишається увімкненим, але екран – сірим.
- `SCREEN BRIGHT WAKE LOCK` – екран блокується з нормальною яскравістю.
- `FULL WAKE LOCK` – блокування клавіатури та екрану зі звичайним підсвічуванням.

Тільки `PARTIAL WAKE LOCK` гарантує, що процесор повністю увімкнутий, решта три типи дозволяють процесору засипати після того, як кнопку живлення пристрою натиснуто. Як і блокування адресного простору прикладних програм, блокування, що надаються `PowerManager`, можуть бути об'єднані з тайм-аутом. Крім того, можна розбудити пристрій під час увімкнення екрана.

Управління пам'яттю пов'язане зі змінами в базовому ядрі для поліпшення використання пам'яті в системах з невеликим обсягом оперативної пам'яті. Обидва механізми управління пам'яттю Anonymous SHared MEMory (ASHMEM) та Physical MEMory (PMEM) додали новий спосіб виділення для ядра. Ashmem може бути використано для розподілу загальної віртуальної пам'яті, а pmem дозволяє розподіляти фізичну пам'ять.

Anonymous SHared MEMory забезпечує названі блоки пам'яті, які може бути розділено між кількома процесами. На відміну від звичайної розподіленої пам'яті, ashmem може бути звільнений ядром. Для використання ashmem, процес відкриває файл `/DEV/ashmem` та виконує функцію `mmap()`.

Physical MEMory (PMEM), наприклад, дозволяє драйверам або бібліотекам виділяти блоки фізично безперервної пам'яті. Цей драйвер було написано для компенсації апаратних обмежень конкретного SoC – The MSM7201A.

Оптимізатор пам'яті *Low memory killer* є стандартним оптимізатором ядра *Linux out of memory killer (oom killer)* і використовує евристичні аналізатори для визначення "шкідливості" процесу, щоб мати можливість припинити його роботу. Вибирають процеси з найбільшою кількістю очок із малим обсягом пам'яті. Така поведінка може бути незручною для користувача, оскільки *oom killer* може закрити поточну прикладну програму користувача, коли наразі існують інші процеси в системі, які не впливають на пам'ять.

Драйвер *lowmemory* починає свою роботу заздалегідь до виникнення критичної ситуації нестачі пам'яті. Драйвер здійснює аналіз процесів та інформує їх про можливість закриття для того, щоб процеси могли зберегти свій стан. Якщо ситуація з пам'яттю погіршується, *lowmemory* починає завершувати процеси.

1.5. API мобільних прикладних програм

Тоді як більшість Android прикладних програм написано на Java мові, є багато відмінностей між API Java та Android API. Основною відмінністю є те, що Android не використовує віртуальну машину Java. Замість неї він використовує дві інших: Dalvik або Android Runtime (ART).

Android платформа не містить віртуальної машини Java (Java VM), на якій виконується Java-байт-код. Замість цього класи Java компілюються у власний формат байт-коду, розроблений спеціально для віртуальної машини Dalvik. На відміну від віртуальних машин Java, які використовують стеки, Dalvik VM є архітектурою на базі регістрів.

Dalvik має деякі специфічні особливості, які відрізняють його від інших стандартних віртуальних машин:

1. VM було розроблено, щоб використовувати менше пам'яті.

2. Пул регістрів було змінено, щоб використовувати тільки 32-бітові індекси для спрощення інтерпретатора.

3. Стандартний Java-байт-код виконує 8-розрядні команди стека. Локальні змінні величини мають бути скопійовані зі стека операндів з окремими інструкціями. Dalvik замість цього використовує свій власний 16-бітний набір команд, який працює безпосередньо на локальних змінних величинах. Локальна змінна величина зазвичай використовує поле чотири біти "віртуального регістра".

4. Оскільки байт-код завантажується віртуальною машиною Dalvik та у зв'язку з певним способом завантаження класів віртуальною машиною Dalvik, то відсутня можливість завантажувати jar-файли. Інший порядок має бути використано для завантаження Android-бібліотеки. До того же зміст базового DEX-файлу має бути скопійовано на карту пам'яті перед завантаженням.

5. Як і у випадку Java SE, Android клас `System` дозволяє витягувати властивості системи. Тим не менш, деякі обов'язкові властивості, визначені за допомогою віртуальної машини Java, не мають ніякого значення або мають інше значення в Android.

Наприклад:

- властивість `Java.version` повертає 0, тому що її не використовують на Android;
- властивість `Java.specification.version` незмінно повертає 0,9, незалежно від версії Android;
- властивість `Java.class.version` незмінно повертає 50, незалежно від версії Android;
- `User.dir` має інше значення на Android;
- `User.home` та `user.name` властивостей не існує в Android.

6. Бібліотека класів Dalvik не збігається ні з Java SE, ні з бібліотекою класів Java ME (наприклад, Java ME класи, AWT або Swing не підтримуються). Замість цього Dalvik використовує свою власну бібліотеку, побудовану на підмножині реалізації Java, – Apache Harmony.

7. Пакет java.lang. Поза вимогою вихідні потоки System.out та System.err не виводять нічого, розробникам рекомендовано використовувати клас `Log`, який записує рядки в `LogCat` (це змінилося, у версії `HoneyComb`, де рядки виводять у лог-консолі).

8. Графіка та бібліотека віджетів. Android не використовує *Abstract Window Toolkit* та бібліотеку *Swing*. Інтерфейс користувача побудовано

з використанням уявлень об'єктів. Android використовує фреймворк, подібний до *Swing*, а не JComponents. Проте Android-віджети не є JavaBeans.

9. Менеджер компонування. На відміну від Swing, де менеджери компонування можуть бути застосовані до будь-якого контейнера-віджета, поведінка уявлення Android кодується безпосередньо в контейнерах.

10. Платформа Android, підтримує відносно велику підмножину бібліотек Java Standard Edition 5.0. Деякі функції було видалено, тому що вони просто не мають сенсу (наприклад, print), а інші є специфічними для Android (наприклад, інтерфейси користувача).

Платформа Android підтримує такі стандартні пакети Java 2 Platform Standard Edition 5.0 API:

- java.io – файлове введення/виведення;
- java.lang (крім java.lang.management) – підтримка мови та винятків;
- java.math – великі числа, округлення, точність;
- java.net – мережа введення/виведення, URL, сокети;
- java.nio – файлове та канальне введення/виведення;
- java.security – авторизація, сертифікати, відкриті ключі;
- java.sql – інтерфейси баз даних;
- java.text – форматування, природна мова, параметри сортування;
- java.util (включаючи java.util.concurrent) – списки, карти, набори, масиви, колекції;
- javax.crypto – шифри, відкриті ключі;
- javax.net – сокет фабрики, SSL;
- javax.security (крім javax.security.auth.kerberos, javax.security.auth.spi та javax.security.sasl);
- javax.sound – музика та звукові ефекти;
- javax.sql (крім javax.sql.rowset) – додаткові інтерфейси баз даних;
- javax.xml.parsers – XML – синтаксичний аналіз;
- org.w3c.dom (але не субпакети) – DOM вузли та елементи;
- org.xml.sax – простий API для XML.

Пакети, які не підтримуються платформою Android:

- java.applet;
- java.awt;
- java.beans;
- java.lang.management;

- java.rmi;
- javax.accessibility;
- javax.activity;
- javax.imageio;
- javax.management;
- javax.naming;
- javax.print;
- javax.rmi;
- javax.security.auth.kerberos;
- javax.security.auth.spi;
- javax.security.sasl;
- javax.swing;
- javax.transaction;
- javax.xml (кроме javax.xml.parsers);
- org.ietf. * ;
- org.omg. * ;
- org.w3c.dom. * (суб-пакети).

По-третє існує ряд сторонніх бібліотек для Android SDK:

- org.apache.commons.codec – утиліти для кодування та декодування;
- org.apache.commons.httpclient – HTTP автентифікація, cookies, методи та протоколи;
- org.bluez – підтримка Bluetooth;
- org.json – формат JavaScript Object Notation.

Запитання для самодіагностики

1. Назвіть рівні програмного забезпечення Android.
2. Охарактеризуйте засоби управління ресурсами мобільних пристроїв ОС Android.
3. Дайте характеристику віртуальної машини Dalvik.
4. Дайте характеристику віртуальної машини ART.
5. Які вам відомі обмеження на використання Java API у мобільному пристрої?
6. Наведіть основні засоби розроблення мобільних додатків.
7. Наведіть основні середовища розроблення IDE.

Розділ 2. Архітектура мобільних додатків

Мета: надати основні принципи щодо архітектури мобільних додатків.

Компетентності: проектувати мобільний інтерфейс користувача; розробляти інтерактивні елементи управління мобільного додатка; визначати ефективні сценарії взаємодії користувача з мобільним пристроєм.

Знати: основні групи візуальних компонентів мобільних платформ; структуру мобільного додатка для мобільних платформ Android і Microsoft Windows 10; основні властивості візуальних компонентів, підходів до проектування адаптивного інтерфейсу користувача та життєвого циклу візуальних компонентів мобільної платформи.

Вміти: проектувати сучасний інтерфейс користувача (UI) мобільного пристрою; програмувати життєвий цикл компонентів мобільного додатка; ефективно формувати комунікаційну стратегію під час роботи в команді.

Нести відповідальність: за якість інтерфейсу користувача та строки його реалізації; за якість реалізації функціональних вимог до мобільного додатка та строків її реалізації.

Основні питання:

1. Огляд архітектурних моделей для розроблення мобільних прикладних програм.
2. Огляд засобів розроблення мобільних прикладних програм.
3. Огляд інтегрованих середовищ розроблення (IDE).
4. Завдання та стек переходів назад (Tasks and Back Stack).
5. Компоненти інтерфейсу.
6. Дослідження життєвого циклу візуальних компонентів.
7. Використання об'єкту Intent та фільтрів.

Ключові слова: активність, сервіс, контент, Model-View-Controler, компонент, шаблон, середовище розроблення.

2.1. Огляд архітектурних моделей для розроблення мобільних прикладних програм

Із кожним роком збільшується частка розробок прикладних програм для мобільних платформ. Однак типові мобільні прикладні програми відрізняються своїм життєвим циклом від настільних прикладних програм.

Мобільні прикладні програми часто потребують більшої взаємодії з користувачем, порівняно з десктопними і веб-програмами, переважно очікуючи дії з боку користувача (наприклад, натискання кнопки), перш ніж відповідати користувачеві оновленим інтерфейсом із зазначенням інформації, яку запитують. У розділі розглянуто модель програмного забезпечення, що фокусується на інтерактивних аспектах мобільних прикладних програм. Також здійснено аналіз загальної концепції розроблення мобільних прикладних програм.

Модель Model View Controller (MVC) є популярним підходом для розроблення мобільних прикладних програм. Слід зауважити, що основна робота більшості мобільних прикладних програм полягає в отриманні даних зі сховища даних та оновлення інтерфейсу користувача з новою запитуваною інформацією на основі запитів користувачів. Таким чином, має сенс пов'язати компоненти інтерфейсу користувача з компонентами сховища даних. Однак, оскільки компоненти інтерфейсу користувача оновлюються частіше, щоб задовольняти змінні вимоги користувачів та нові технології, ніж компоненти сховища даних, то необхідно ввести додаткові компоненти. Метою такої схеми є розподіл компонентів на компоненти інтерфейсу користувача (View), компоненти, які забезпечують основні функціональні можливості (Controller) та дані (Model).

Як було сказано раніше, основними компонентами Android-прикладних програм є:

1. Активність – становить єдиний клас інтерфейсу користувача.

2. Сервіс – пов'язаний із завданнями, які будуть виконувати у фоновому режимі потоків (наприклад, мережевих операцій), не торкаючи до того ж компоненти інтерфейсу користувача.

3. Постачальник контенту – дозволяє зберігати дані у прикладній програмі з використанням SQLite бази даних або SharedPreferences (дані, що зберігаються у файлі XML на пристрої).

4. Широкомовний приймач – відповідає на повідомлення системи (наприклад, попередження про низький рівень заряду) та забезпечує повідомлення користувача.

Зазвичай стандартний проект мобільної прикладної програми містить такий набір компонентів:

компоненти інтерфейсу користувача;

сервісні компоненти (місце розташування пристрою для відстеження місця розташування користувача за допомогою GPS, фонові завдання для отримання даних з інших служб та ін.);

компоненти даних (для зберігання та вилучення налаштувань);

віджет-компоненти (кнопки, поля для редагування тексту, поля для перегляду тексту, оброблення кліків користувачів і т. д.).

Компоненти для інтерфейсу користувача складаються з пакета Активностей, які за своєю сутністю нагадують компонент View моделі MVC. Кожна Активність оновлює та модифікує призначений для користувача інтерфейс протягом усього життєвого циклу програми. Компоненти послуг та даних схожі на компонент Model у тому, що вони спостерігають за поведінкою даних.

Вони також обробляють запити від View і Controller для отримання інформації про їхній стан та інструкції, як змінити їх стан. Компоненти віджета аналогічні Controller у тому, що очікують уведення користувача (наприклад, натискання кнопки), а потім інтерпретують ці входи та оповіщають Model або View для того, щоб внести зміни.

Таким чином, можна зробити висновок, що на рівні стандартної мобільної прикладної програми модель MVC добре вписується в Android системи та кожен компонент програми більш-менш має своє відображення в Model, View і Controller. Однак за найближчого розгляду деталей усе не так просто. Прикладом може бути компонент Activities, який є складовою частиною програми. Кожна Активність складається з Java-файла і відповідного файла макета XML. Розробник визначає розташування UI-активності у файлі XML та реалізує функціональність і поведінку у файлі Java (розподіл інтерфейсу користувача з логікою прикладної програми). Коли використовують віджет-компоненти необхідно спочатку визначити віджети як елементи XML-файла макета для створення екземпляра віджета та його атрибутів. Оброблення поведінки віджета у відповідь на дії користувача також реалізують в файлі Java. Проте розробник має доступ до атрибутів віджета з макета XML під час реалізації його функціональних можливостей, і це порушує принцип розподілу View із Model/Controller шаблона MVC.

Можливе вирішення цієї проблеми полягає у використанні шаблона Observer (Спостерігач) у MVC. Можна реалізувати інтерфейс Observer у компоненті Активності (View), щоб отримати оновлення та повідомлення

про зміни в логіці програми (стан Model). Коли відбувається зміна, логіка прикладної програми повідомляє всі Активності (спостерігачів) про зміну стану. Це виключає зв'язок між моделлю та уявленням (рис. 2.1).

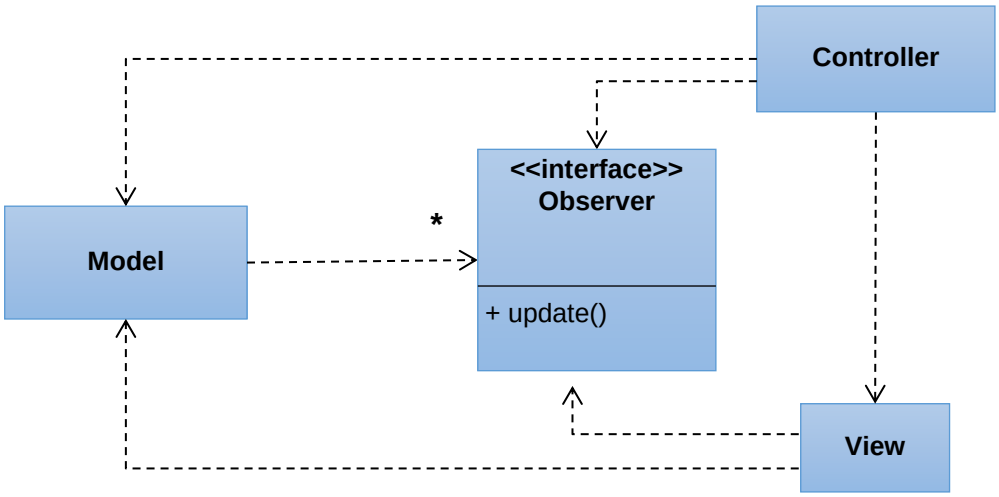

Рис. 2.1. **Використання Observer у межах MVC**

Шаблон Layered Abstraction складається з ієрархії рівнів. На кожному шарі наявні компоненти, які працюють разом у межах того ж рівня абстракції, із рівнем забезпечення функціональних можливостей нижчим для вищого шару.

Android-архітектура дотримується шаблона Layered Abstraction (рис. 2.2). Вона складається з чотирьох шарів абстракції із прямим зв'язком тільки між поточним шаром і шаром безпосередньо над або під ним:

1. Область застосування – цей шар складається з набору основних прикладних програм, а також розроблених прикладних програм.

2. Каркас прикладної програми – шар, який містить основні компоненти системи (Android діяльності, послуги, контент-провайдер, широкомовний приймач), а також інші компоненти системи.

3. Бібліотеки – складається з основних бібліотек Android системи, а також реляційної системи управління базами даних – SQLite.

4. Ядро Linux – містить драйвери пристроїв, які забезпечують зв'язок з апаратним забезпеченням пристрою.

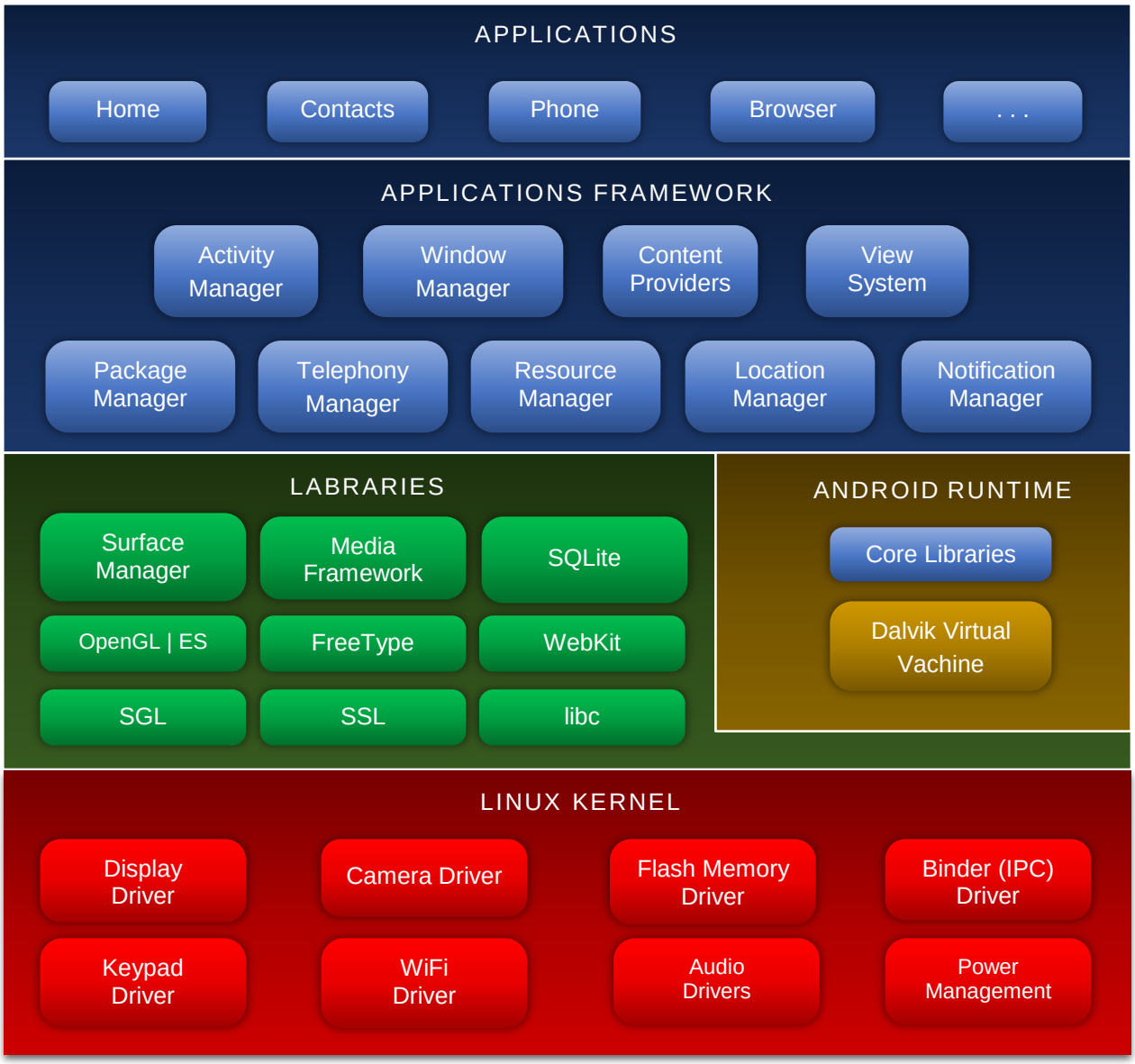

Рис. 2.2. **Архітектура системи Android**

Із точки зору розробника прикладних програм, із великою кількістю компонентів для інтерфейсу користувача було б доцільно використовувати шаблон MVC. Використання цього шаблона дає можливість зменшити кількість програмного коду, зберігаючи до того ж ядро бізнес-логіки прикладної програми, у разі зміни вимог користувача.

За допомогою шаблона Layered Abstraction, який є базовою моделлю для Android системи, можна будувати стандартні мобільні прикладні програми, які відсилають запити на роботу із сервісами та отримують повідомлення від них.

2.2. Огляд засобів розроблення мобільних прикладних програм

Комплект розроблення програмного забезпечення Android (SDK), містить повний набір інструментів розробника: відладчик; бібліотеки; емулятор мобільного пристрою, заснований на QEMU; документацію; зразки коду та підручники [38]. У цей час підтримують платформи розроблення які працюють на операційній системі Linux (будь-який сучасний робочий стіл Linux), Mac OS X 10.5.8 або пізнішої версії та Windows XP або більш пізньої версії. Станом на березень 2015 року SDK не доступна на Android, але розроблення програмного забезпечення можливе за допомогою спеціалізованих програм для Android (табл. 2.1) [10; 25; 26].

Таблиця 2.1

SDK Android

Розробник(и)	Google
Перший випуск	жовтень 2009 року
Стабільна версія	2.1.3, на 15 серпня 2016
Мова програмування	Java
Операційна система	Крос-платформна
Мова	Англійська
Тип	IDE, SDK
Сайт	developer.android.com/tools/sdk/eclipse-adt.html, developer.android.com/sdk/index.html

Приблизно до кінця 2014 року офіційно підтримуваною інтегрованим середовищем розробкою (IDE) була Eclipse (рис. 2.3), що пропонувала для розроблення мобільних прикладних програм спеціальний плагін – інструменти розроблення Android (ADT), хоча середовище IntelliJ IDEA (усі випуски) повністю підтримує Android-розроблення з коробки (рис. 2.4) [37]. NetBeans IDE також підтримує Android-розроблення за допомогою плагіна (рис. 2.5) [30].

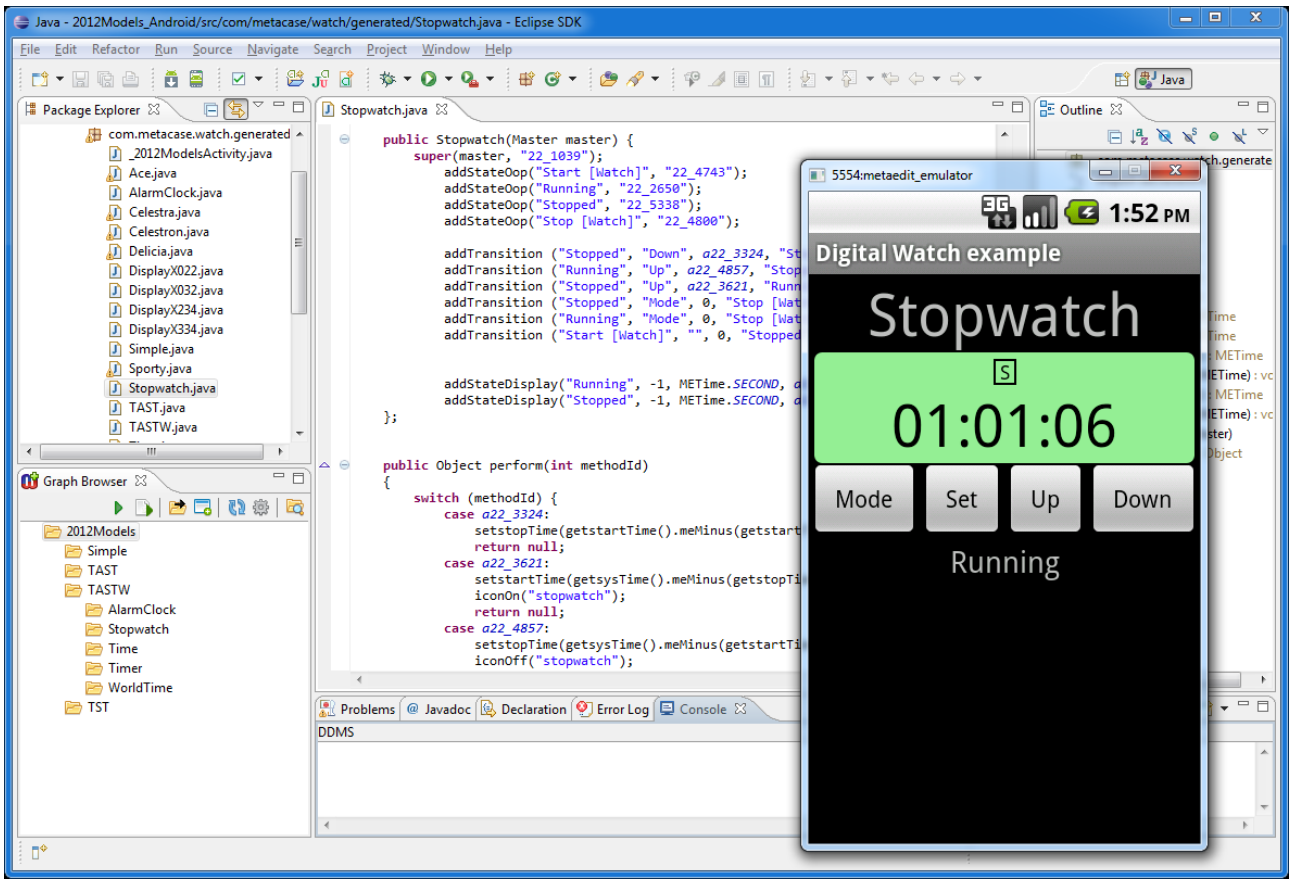

Рис. 2.3. **IDE Eclipse**

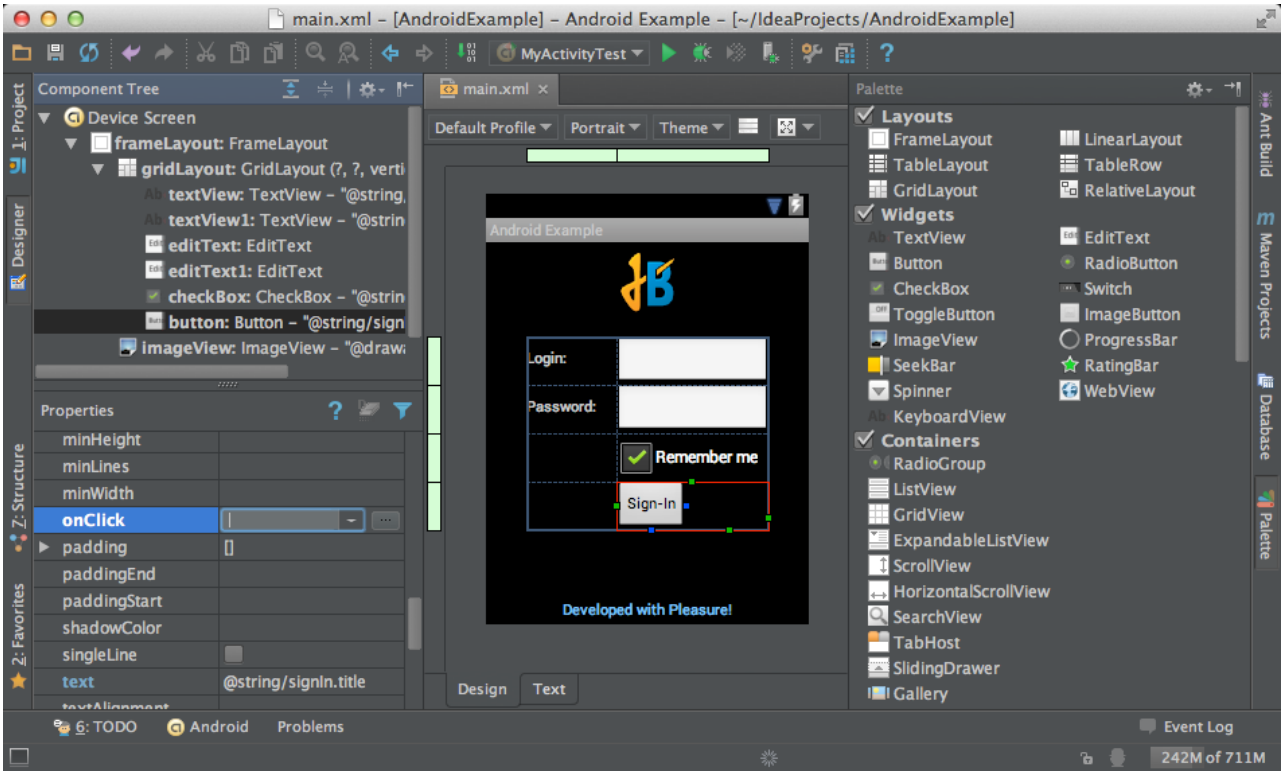

Рис. 2.4. **IntelliJ IDEA**

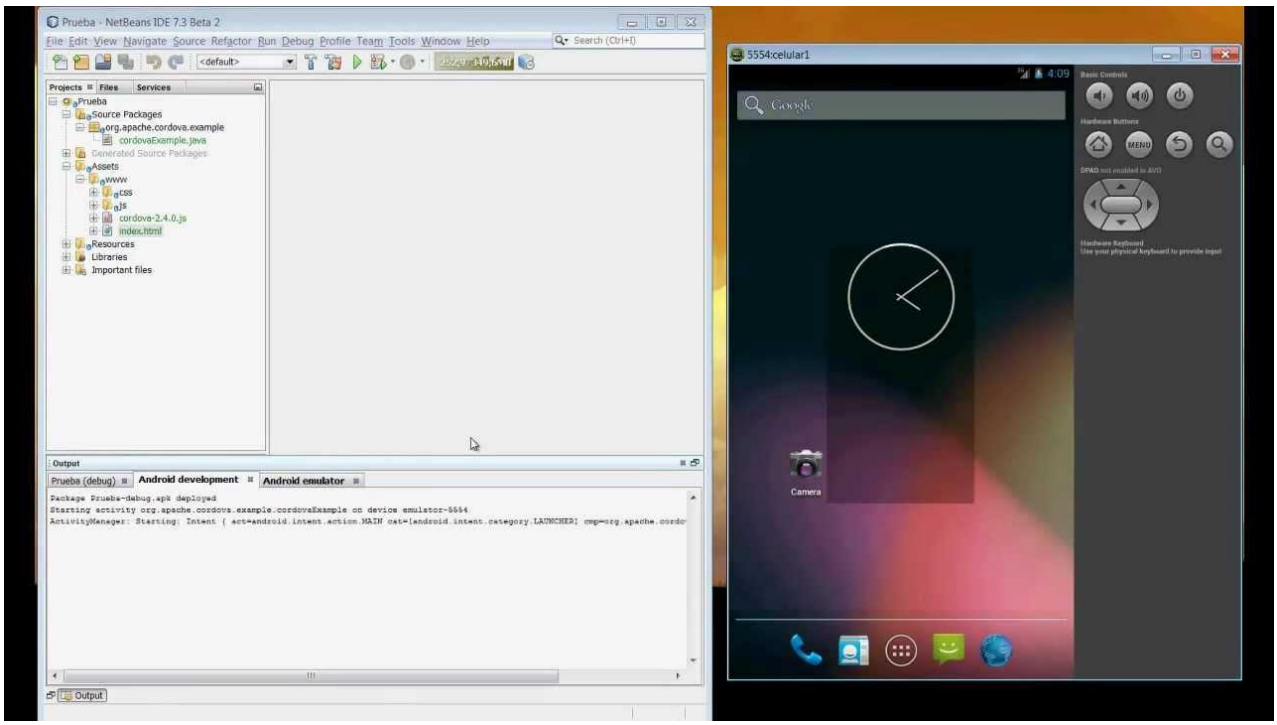

Рис. 2.5. **NetBeans IDE**

У 2015 році Google спільно з IntelliJ презентували офіційну IDE від Google Android Studio [14]. Однак розробники можуть використовувати інші інтегровані середовища розроблення. Крім того, розробники можуть використовувати будь-який текстовий редактор для редагування XML та Java-файлів, а потім використовувати інструменти командного рядка (комплект розроблення Java і Apache Ant), щоб створити, скомпілювати та налагодити прикладні програми Android, а також управляти підключеними пристроями Android (наприклад, викликавши перезавантаження, установлення програмного забезпечення, видалення пакетів) [19].

Поліпшення у SDK Android ідуть пліч-о-пліч із загальним розвитком Android-платформи. SDK також підтримує старі версії Android-платформи, якщо розробники хочуть запускати свої прикладні програми на старих пристроях. Засоби розроблення є компонентами, що завантажуються так, що після завантаження останньої версії та платформи старі платформи та інструменти розроблення також можуть бути завантажені для тестування сумісності [33].

Додатки Android упаковують у файли у форматі .apk та зберігають у папці /data /app на пристрої Android (папка доступна тільки для суперкористувача з міркувань безпеки). Пакети apk містять файли .dex [20] (байт-код, що виконує прикладна програма, для Dalvik VM), файли ресурсів тощо.

ADB (Android Debug Bridge). Клієнт-серверна прикладна програма, яка надає доступ до працюючого емулятора або пристрою (рис. 2.6). Із його допомогою можна копіювати файли, установлювати скомпільовані програмні пакети та запускати консольні команди. Використовуючи консоль, можна змінювати налаштування журналу та взаємодіяти з базами даних SQLite, які зберігаються на пристрої. У старих версіях SDK програма перебувала в папці tools, тепер вона знаходиться в папці platform-tools.

Складається із трьох компонентів: фонової служби (демону), що працює в емуляторі; сервісу, запущеному на комп'ютері розробника та клієнтської програми (на зразок DDMS), яка зв'язується зі службою через сервіс.

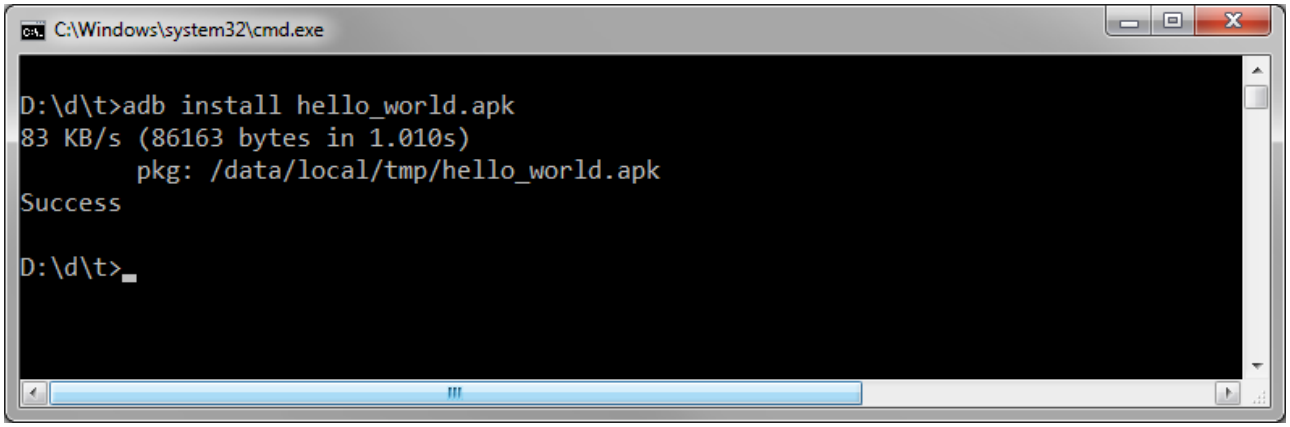

Рис. 2.6. **Установлення пакета apk з використанням adb**

Fastboot. Fastboot є діагностичним протоколом, який іде в комплекті із SDK та використовується, у першу чергу, для зміни флеш файлової системи через USB-з'єднання з комп'ютером. Fastboot потребує, щоб пристрій запускався завантажувачем у режимі Second Program Loader, у якому виконують тільки найосновніші ініціалізації обладнання. Після увімкнення протоколу на самому пристрої воно буде приймати певний набір команд, надісланих до нього через USB, використовуючи командний рядок.

Деякі з найбільш часто використовуваних команд швидкого завантаження:

flash – переписує розділ із двійковим зображенням, що зберігається на комп'ютері;

erase – видаляє певний розділ;

reboot – перезавантаження пристрою або в основну операційну систему, або назад у завантажувач;

devices – відображає список усіх пристроїв (із серійним номером), підключених до комп'ютера;

format – форматує певний розділ; файлова система розділу має розпізнаватися пристроєм.

Android NDK. Android Native Development Kit (NDK Android). Бібліотеки, написані на C, C++ та інших мовах може бути скомпільовано в ARM, MIPS або x86 машинний код та встановлено на пристрій із використанням набору Android Native Development Kit (NDK). Рідні класи може бути викликано з Java-коду, що виконується під Dalvik VM, використовуючи виклик System.loadLibrary, який є частиною стандартних класів Java в Android (табл. 2.2) [12; 29].

Таблиця 2.2

Android Native Development Kit (NDK Android)

Розробник(и)	Google
Перший випуск	червень 2009 року
Стабільна версія	24.4.1, жовтень 2015
Мова програмування	C та C++
Операційна система	Крос-платформена
Доступна	Англійська
Тип	IDE, SDK
Сайт	developer.android.com/tools/sdk/ndk/index.html

Розроблені прикладні програми може бути скомпільовано та встановлено за допомогою традиційних інструментів розроблення [13]. Тим не менш, відповідно до Android-документації, NDK не мають використовувати виключно для розроблення прикладних програм тільки тому, що розробник уважає за краще програмувати на C/C++, оскільки використання NDK збільшує складність прикладної програми, що не піде йому на користь.

ADB-відладчик дає права root в Android Emulator, що дозволяє завантажувати та виконувати програмний код оптимізований під ARM, MIPS або x86 процесори. Машинний код може бути скомпільовано з використанням GCC або Intel C++ Compiler на стандартному ПК. Запуск машинного коду на Android платформі ускладнено використанням нестандартної

бібліотеки C (Libc, відомої як Bionic). Графічну бібліотеку Android, яку використовують для арбітражу та контролю за доступом до цього пристрою названо Graphics Library Skia (SGL), її випущено під відкритою ліцензією. Skia має движки для обох Win32 та Unix-платформ, дозволяючи розвивати крос-платформні прикладні програми. Skia також має графічний движок, що лежить в основі веб-браузера Google Chrome [34].

На відміну від розроблення додатків Java, заснованих на використанні таких IDE, як Eclipse, NDK засновано на командному рядку, він потребує введення команд уручну для компілювання, розгортання та налагодження прикладних програм. Деякі інструменти сторонніх розробників дозволяють інтегрувати NDK в Eclipse та Visual Studio.

Платформа ADK (Android Accessory Development Kit). ADK – це пристрій, що підтримує Android Open Accessory Protocol. ADK – це Arduino-сумісна платформа, що підключається до Android-пристрою через USB або Bluetooth та містить безліч датчиків, сенсорів та індикаторів.

Google пропонує два напрями застосування ADK:

комерційний – аудіо док-станції, інтеграція в спортивні тренажери та ін.;

хобі – контролери роботизованої техніки.

ADK 2012 містить:

Arduino-сумісну плату на основі ARM 32-bit Cortex M3 мікропроцесора;

два USB. Один для підключення до Android пристрою, інший – для налагодження та програмування;

датчики світла, кольору, наближення, температури, вологості, атмосферного тиску та акселерометр;

слот для SD карти;

підтримку Bluetooth;

шість семисегментних світлодіодних RGB-матриць. 12 "party mode" світлодіодів;

ємнісний слайдер (гучність, яскравість і т. д.) та кнопки – по дві на кожну цифру та додатково ще вісім;

підсилювач звуку та динамік;

NFC-мітку, доступну на запис.

Платформа дає можливість відтворювати аудіо з Android-пристрою по USB-з'єднанню. Вимоги – Android 4.1 (API Level 16 та вищий).

Сам комплект ADK 2012 має вигляд закінченого пристрою у формі будильника та функцією аудіо дока (рис. 2.7).

Рис. 2.7. **ADK 2012**

Убудована підтримка Go. Починаючи з версії 2.4 для програмістів, які пишуть Android-програми на мові програмування Go, включено підтримку мови без будь-якого Java-коду, хоча й з обмеженим набором інтерфейсів Android.

Android APIMiner. Android-APIMiner – це платформа, яка є інструментом автоматичної генерації та вилучення документації Javadoc із реальних прикладних програм Android із відкритим вихідним кодом та прикладами використання (рис. 2.8). Для поліпшення якості витягнутих прикладів, APIMiner використовує внутрішньо-процесуальний статичний алгоритм витягування [21].

У табл. 2.3 надано опис основних компонентів Android APIMiner.

Таблиця 2.3

Компоненти Android APIMiner

Компоненти	Описи
1	2
Source Code Analyzer	Цей модуль аналізує вихідний код API та клієнтських систем для витягування структурних даних
Patterns Analyzer	Цей модуль витягує шаблони використання елементів API на основі їхнього використання

1	2
Examples Extractor	Цей модуль витягує приклади використання з вихідного коду клієнтських систем
Recommendation Engine	Цей модуль генерує рекомендації шаблонів та приклади використання на основі даних, витягнутих за допомогою попередніх модулів
JavaDoc Weaver	Цей модуль генерує документацію API, включаючи приклади використання

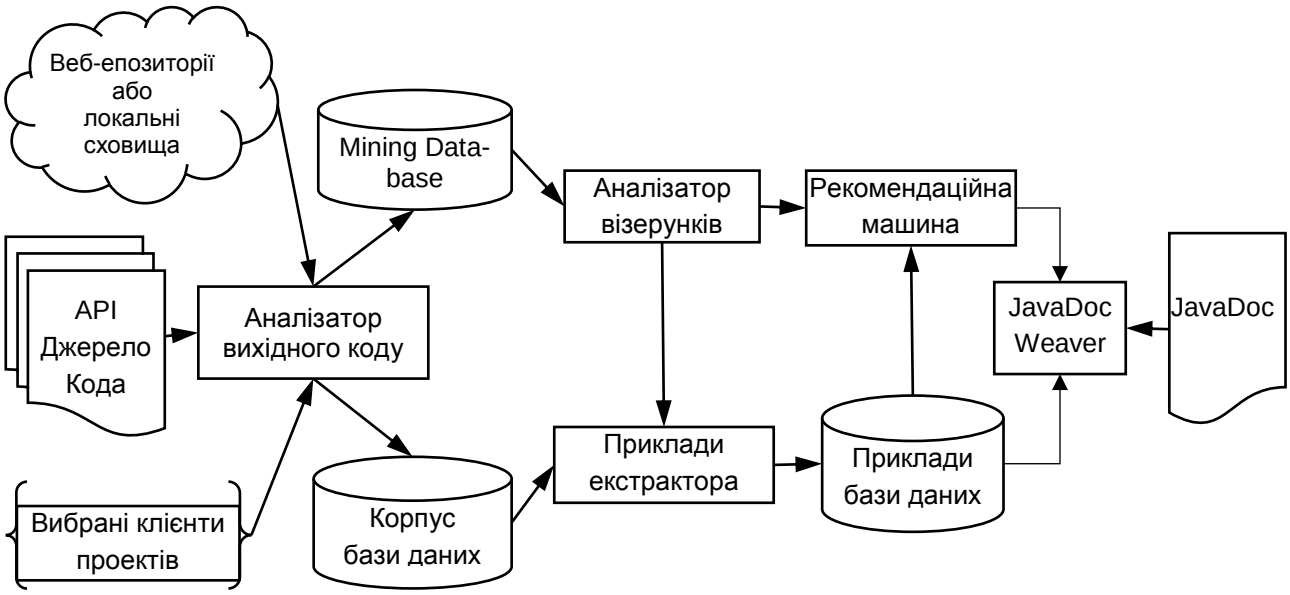

Рис. 2.8. **Компоненти Android APIMiner**

AndroWish. Tcl (Tool Command Language) є дуже потужною, але легко досліджуваною динамічною мовою програмування, яка підходить для дуже широкого спектру застосування, у тому числі для мережевих і настільних прикладних програм, мережевого програмування, тестування та багато чого іншого. Tcl має відкритий вихідний код і дійсно є кросплатформною, що легко розгортається та розширюється.

Tk – інструментарій для створення графічного інтерфейсу користувача піднімає розроблення настільних прикладних програм на більш високий рівень, ніж звичайні підходи. Tk – є стандартом інтерфейсу не тільки для Tcl, але й для багатьох інших динамічних мов, та дозволяє створювати насичені програми, які працюють без змін під ОС Windows, Mac OS X, Linux.

AndroWish дозволяє запускати настільні Tcl і Tk програми майже в незмінному вигляді на Android-платформі [15].

Деякі особливості AndroWish наведено далі.

1. Рідний Tcl/Tk 8.6 порт для платформи Android (починаючи з версії 3.3.3 або вищої) для процесорів ARM і x86.

2. Виконання наявних Tcl/Tk скриптів на Android-платформі без змін.

3. За основу взято ранній проект Тіма Бейкера SDLTk.

4. Емуляції X11 на основі AGG (Anti-Grain-Geometry) та SDL 3.0.

5. Забезпечує згладжування ліній, кіл, дуг.

6. Візуалізація шрифтів із використанням движка шрифтів Freetype.

7. Містить віджет 3D-полотна, який використовує OpenGL для емуляції малювання OpenGLES 2.2.

8. Містить віджет tkpath, який розширює такі можливості канви SVG, як: рендеринг, альфа-канали, підтримка TrueType контурних шрифтів.

9. Деякі конкретні об'єкти Android доступні через SDL та їх викликають командою sdltk.

10. Використання AndroWish потребує Android SDK та Android NDK.

11. Тестування та налагодження сценаріїв Tcl на Android-пристроях можна здійснювати із системи розроблення з використанням tkconclient. Файли може бути передано за допомогою підключення SSH/SFTP, як описано у tkconclient.

12. Експериментальна функція дозволяє будувати невеликі прикладні програми, які використовує як основу встановлений AndroWish.

App Inventor для Android. 12 липня 2010 року Google оголосила про доступність App Inventor для Android (рис. 2.9). App Inventor – це веб-орієнтоване візуальне середовище розроблення для початківців програмістів, засноване на бібліотеці Open Blocks Java Массачусетського технологічного інституту (MIT). Середовище забезпечує доступ до GPS, акселерометра, даних позиціонування пристрою, телефонних функцій, обміну текстовими повідомленнями, перетворення мови на текст, контактів, даних постійного зберігання та веб-служб.

Останню версію, що була створена в результаті співробітництва Google та MIT, випущено в лютому 2012 року, тоді як першу версію, створену виключно MIT, було запущено в березні 2012 року та оновлено до App Inventor 2 у грудні 2013 року. Із 2014 року App Inventor підтримується виключно MIT [17].

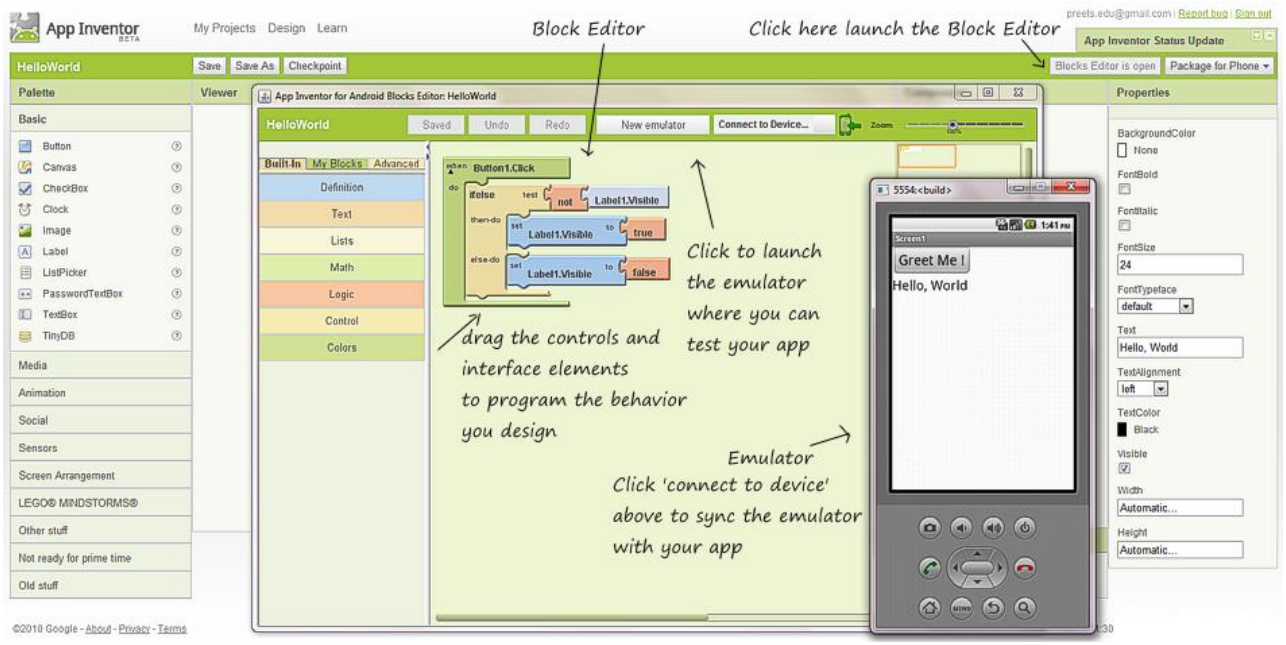

Рис. 2.9. **App Inventor**

Basic4android – VisualBasic Native Android Language. Basic4-Android – це простий і потужний інструмент розроблення прикладних програм для пристроїв, що працюють під управлінням операційної системи Android. Мова Basic4Android (рис. 2.10) дуже схожа на популярну мову Visual Basic. У процесі розроблення прикладних програм використано безліч різних додаткових бібліотек. Для виконання створених програм ніяких додаткових runtime-засобів не потрібно.

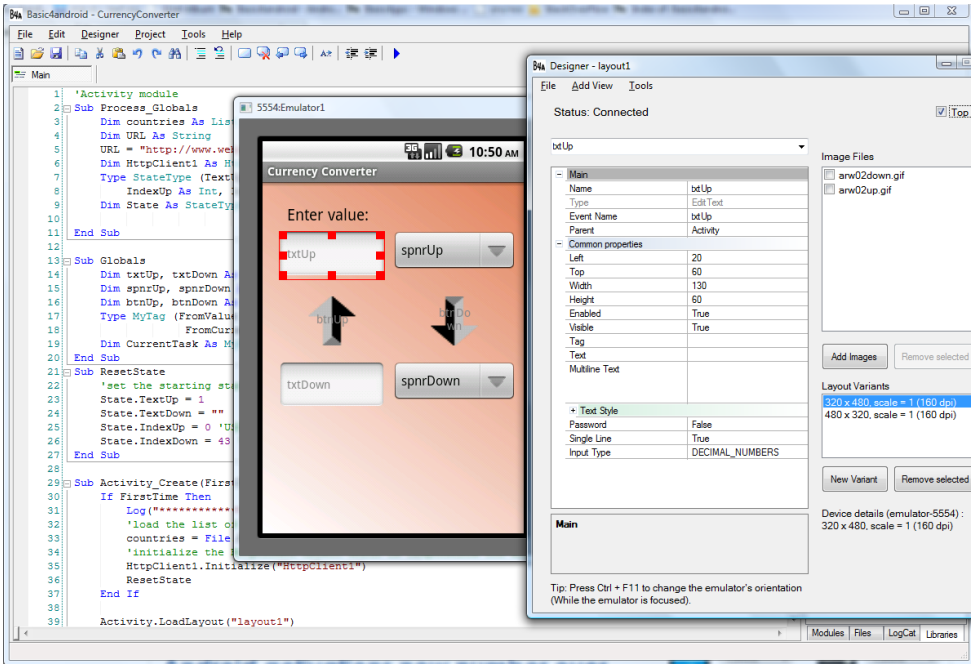

Рис. 2.10. **Basic4android**

Corona SDK. Corona SDK є комплектом розроблення програмного забезпечення (SDK), створеним Вальтером Лухом, засновником Corona Labs Inc. Corona SDK дозволяє програмістам створювати мобільні прикладні програми для iPhone, IPAD та Android-пристроїв (рис. 2.11).

Corona дозволяє розробникам створювати графічні прикладні програми, використовуючи інтегровану Lua-мову, що нашаровується поверх C++/OpenGL. SDK поширюють на основі моделі продажу за передплатою, вона не потребує яких-небудь відрахувань від продажу розроблених прикладних програм та не нав'язує ніяких вимог брендингу.

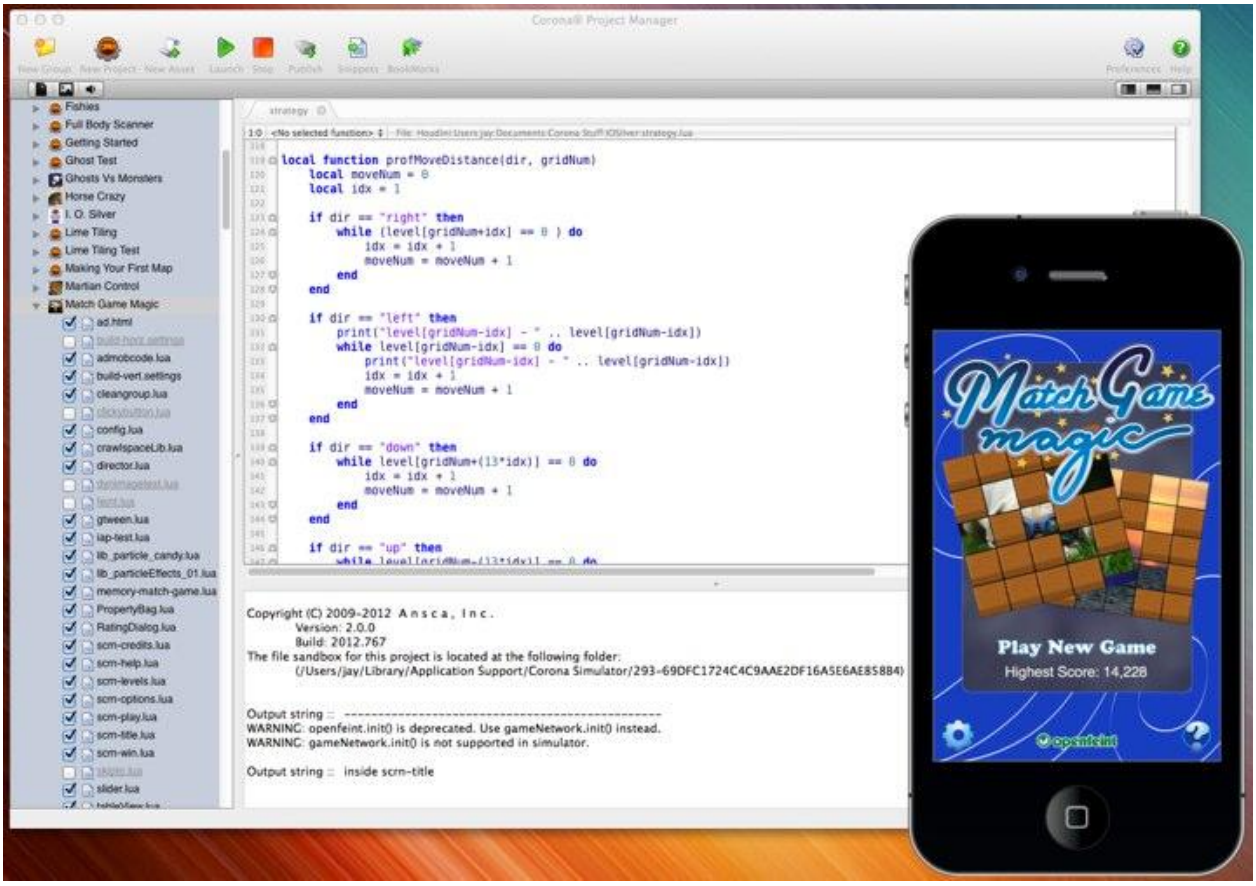

Рис. 2.11. **Corona SDK**

Delphi також може бути використано для створення Android прикладних програм із використанням мови Object Pascal. Останню версію Delphi XE8 розроблено Embarcadero Studio.

Embarcadero® RAD Studio XE8 (рис. 2.12) є закінченим рішенням для розроблення програмного забезпечення для Windows, Mac, IOS, Android та IoT. RAD Studio дозволяє будувати готові рішення, які розробляють не тільки для клієнтських платформ, але також і для мобільних пристроїв, таких смарт-пристроїв, як смарт-годинник та інші гаджети IoT [32].

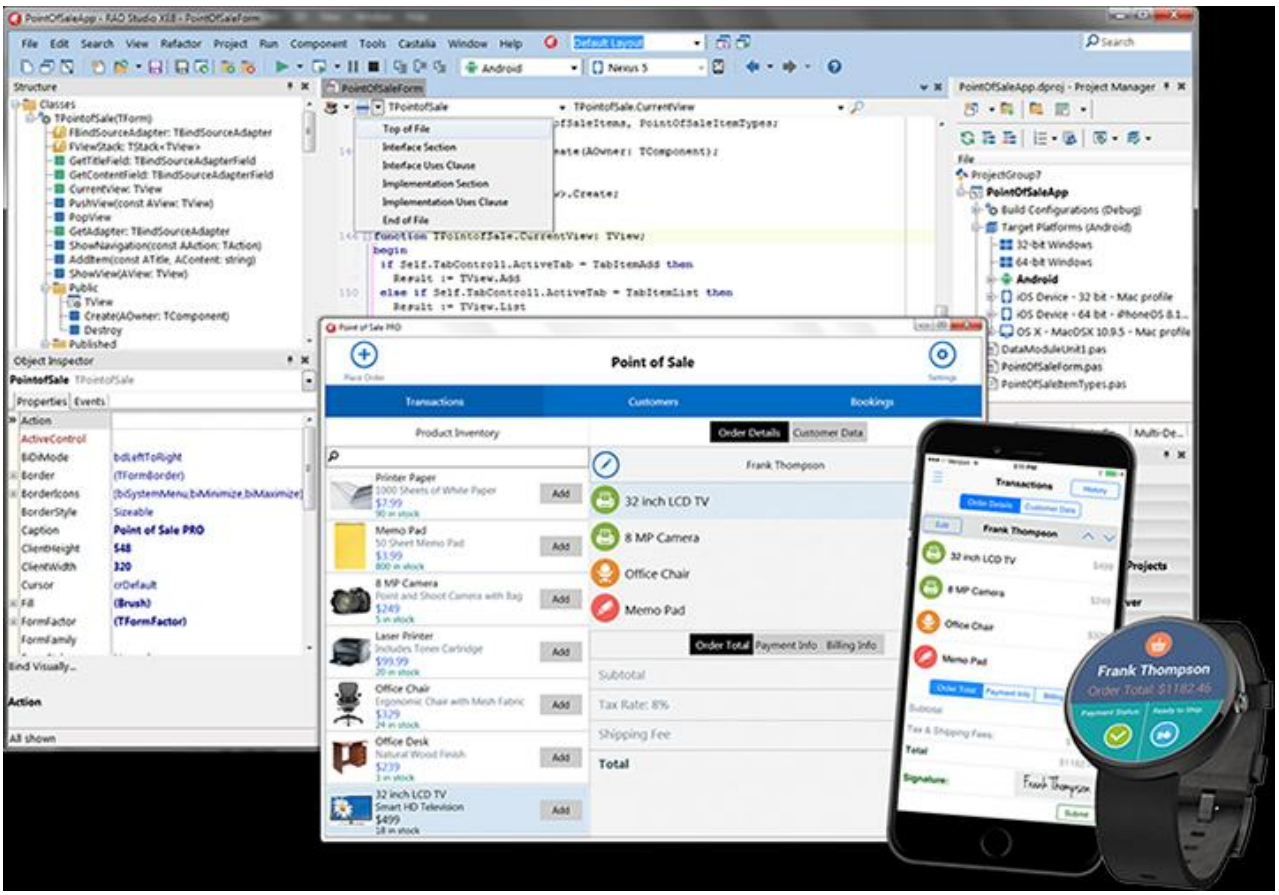

Рис. 2.12. **RAD Studio XE8**

HyperNext Android Creator. HyperNext Android Creator (HAC) є системою розроблення програмного забезпечення, орієнтованою на програмістів-початківців [42], яка допомагає їм створювати свої власні прикладні програми для Android, не знаючи Java та Android SDK (рис. 2.13). HAC засновано на тому, що HyperCard обробляє програмне забезпечення у вигляді купки карток, де в будь-який момент часу видно тільки одну карту, що дуже добре підходить для прикладних програм мобільних телефонів, у яких відображається одноразово тільки одне вікно.

Основною мовою програмування HyperNext Android Creator є Hyper Next, створена на основі мови HyperTalk Hypercard. HyperNext схожа на англійську мову та має багато особливостей, які дозволяють створювати прикладні програми для Android. Вона підтримується в щораз більшій підмножині Android SDK. HAC має власні версії типів управління GUI та автоматично запускає свою власну фонову службу, так що прикладні програми можуть продовжувати працювати та обробляти інформацію у фоновому режимі.

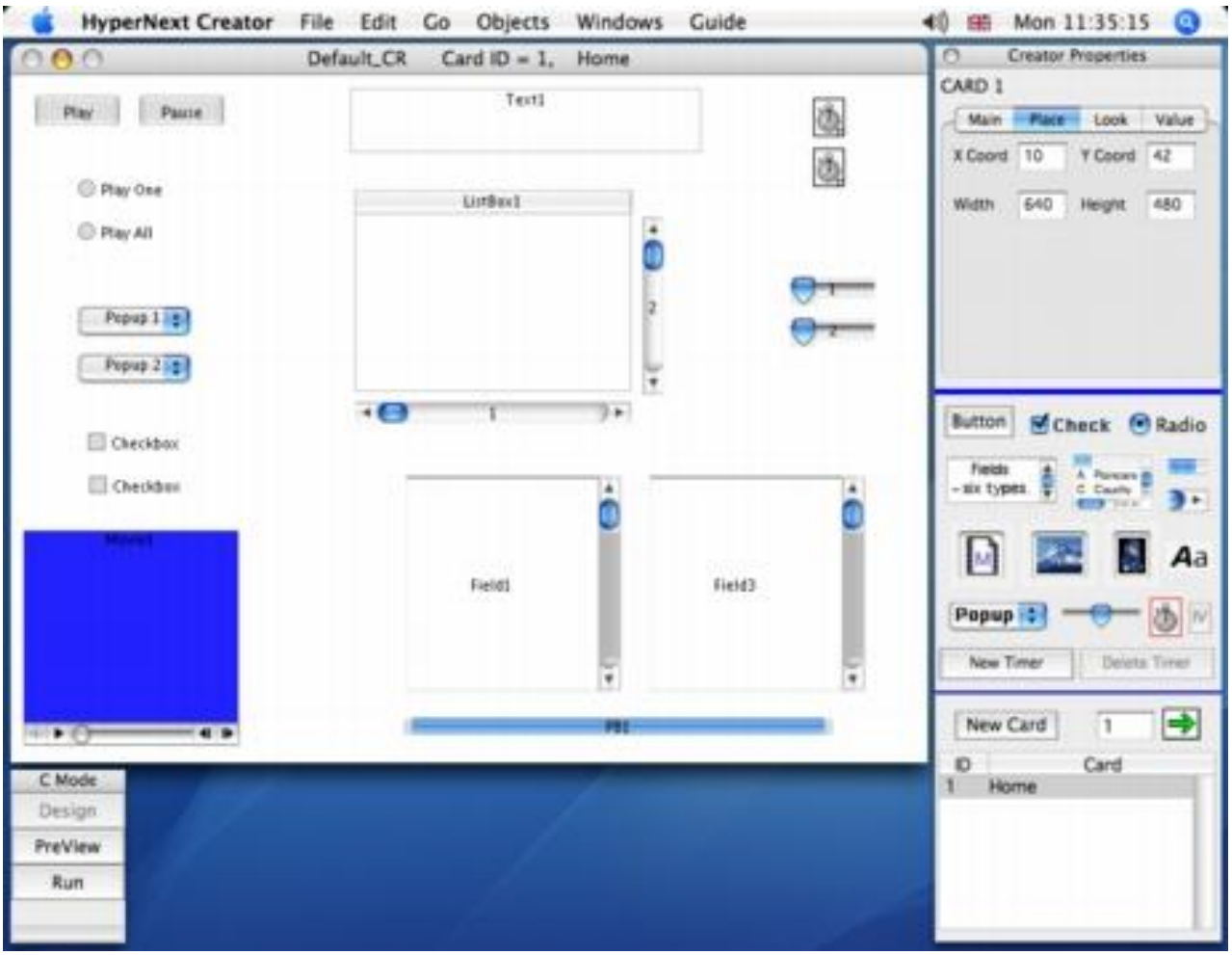

Рис. 2.13. **HyperNext Android Creator**

Kivy – це відкрита бібліотека мовою Python для розроблення прикладного програмного забезпечення мультитач із природним інтерфейсом користувача (NUI) для широкого набору пристроїв (рис. 2.14). Kivy забезпечує можливість підтримки єдиної прикладної програми для численних операційних систем ("кодуй один раз, запускай скрізь"). Kivy має виготовлений на замовлення інструмент розгортання для розгортання мобільних прикладних програм під назвою Buildozer, він доступний тільки для Linux. Buildozer у цей час існує в альфа-версії, але набагато зручніший, ніж громіздкі старі методи розгортання Kivy. Прикладні програми, запрограмовані Kivy, може бути подано на будь-якій мобільній платформі Android-прикладних програм.

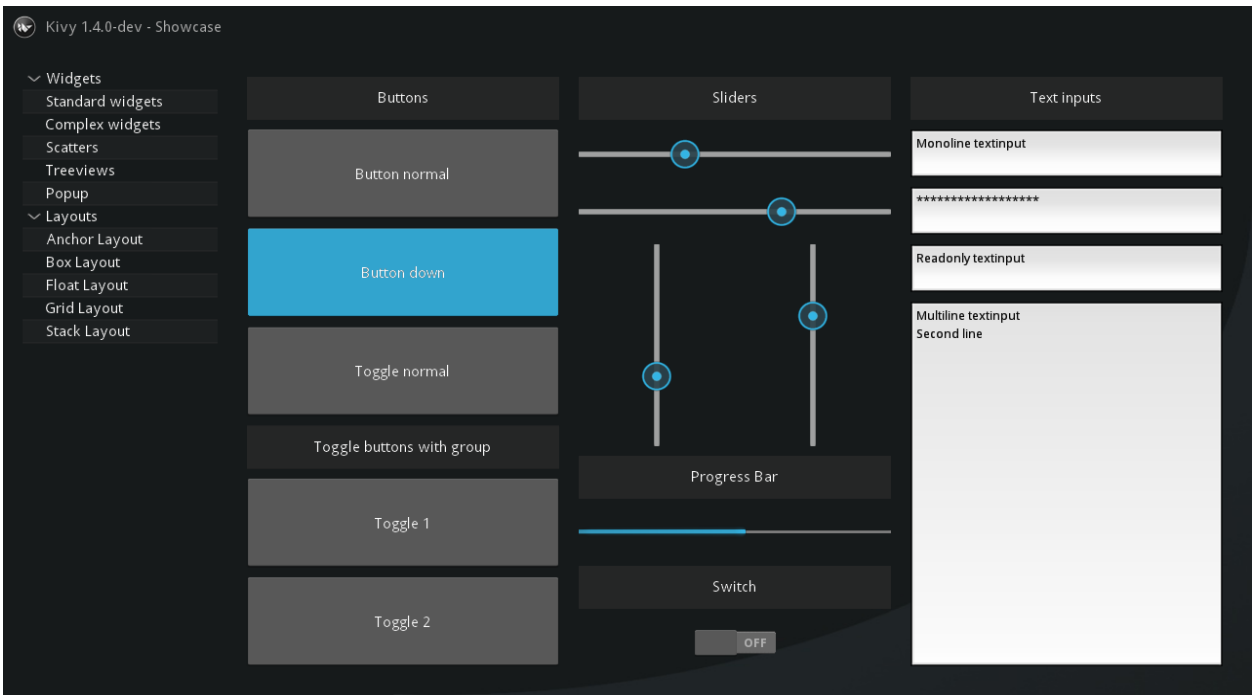

Рис. 2.14. **Kivy 2.4. on Android**

Lazarus можна використовувати для розроблення прикладних програм Android, на мові Pascal із компілятором Free Pascal, починаючи з версії 3.7.2. (рис. 2.15).

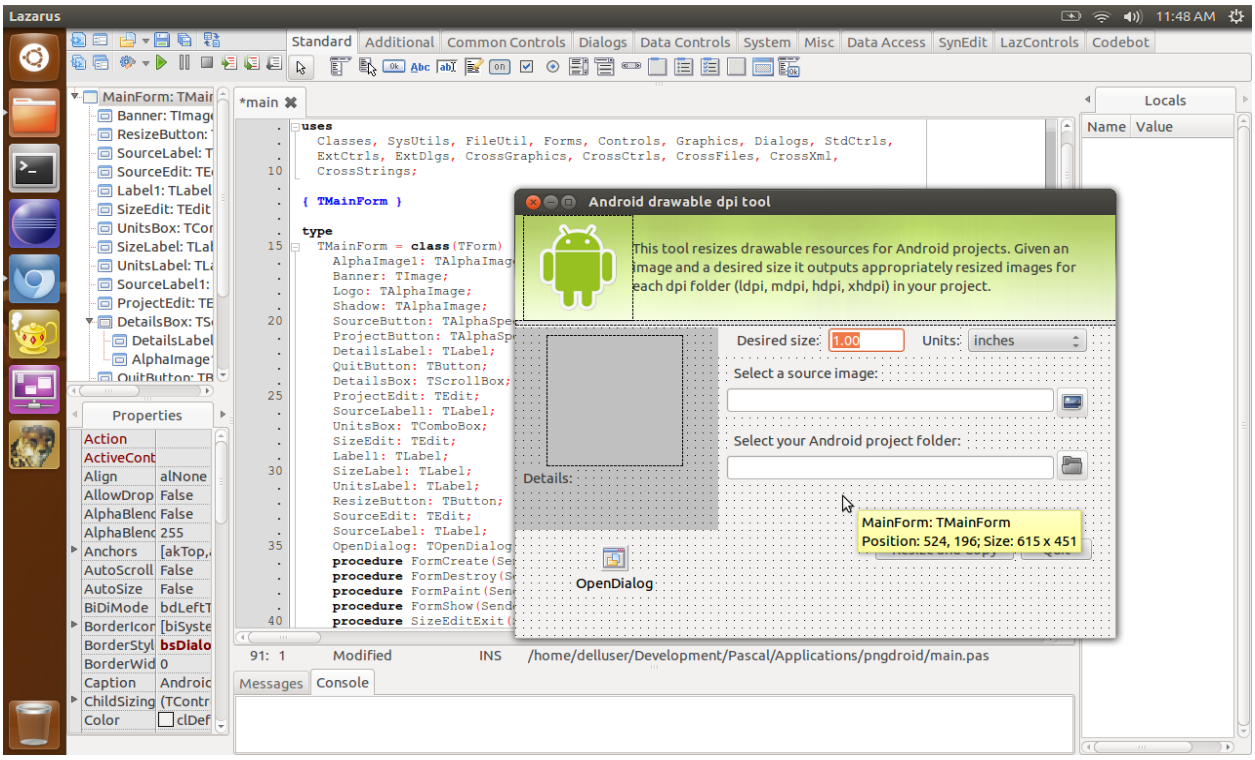

Рис. 2.15. **Проект Lazarus**

Qt для Android, починаючи з версії Qt 5, створює прикладні програми для запуску на пристроях з Android v3.3.3 (рівень API 10) або більш пізньої версії. Qt є основою для крос-платформних прикладних програм, які можна запускати на таких цільових платформах, як: Android, Linux, IOS, Sailfish OS і Windows. Розроблення Qt прикладних програм виконується з використанням мови C++ та QML, потребуючи встановлених Android NDK та SDK. Qt Creator є інтегрованим середовищем розроблення та спільно з Qt Framework його використовують для розроблення мультиплатформних прикладних програм.

RFO BASIC. Це діалект Dartmouth Basic, що є інтерпретатором із набором бібліотек для доступу до апаратного обладнання, датчиків, звуку, графіці, мультитачу, файлової системи, SQLite, мережі, HTML інтерфейсу, шифрування, SMS, функцій телефону, електронної пошти, перетворення тексту на мову, розпізнавання голосу, GPS та інших функцій. Це програмне забезпечення з відкритим вихідним кодом може компілювати автономні APK-файли. RFO Basic активно розвивається з березня 2015.

RubyMotion є набором інструментів для створення мобільних прикладних програм на мові Ruby. Підтримка Android з'явилася у версії RubyMotion 3.0. Прикладні програми Android, створені з використанням RubyMotion, можна назвати загалом набором Java API від Ruby, до того ж можливе використання сторонніх бібліотек Java.

Saphir є відгалуженням із відкритим вихідним кодом від проекту Rebol3 (R3). Уся функціональність R3, у тому числі GUI, графіка, доступ до мережі, доступ до файлів, парсинг та інші особливості портують на основні портативні ОС Android, Windows, Mac, Linux без будь-яких змін у вихідному коді. Saphir дозволяє використовувати шаблони діалектних моделей (DSL) для побудови графічних інтерфейсів користувача та виконання спільних обчислювальних операцій. Невеликий розмір компілятора (0,5 – 1,5 мегабайт) доповнено простим утилітарним дизайном Saphir.

Бібліотека SDL пропонує, крім можливості розроблення з використанням Java, можливість розроблення із використанням C з наступним простим перенесенням наявних SDL та власних прикладних програм C. Застосування Java-ін'єкцій і прокладок JNI дозволяє використовувати рідну бібліотеку SDL під час портування на пристрої Android, наприклад, як у відео грі Jagged Alliance 3.

Метою **The Simple project** є забезпечення розробника простою для розуміння та використання мовою для розроблення прикладних програм для платформи Android [31]. The Simple project є основним діалектом

для розроблення прикладних програм Android. Він спрямований на професійних і непрофесійних програмістів та дозволяє програмістам швидко створювати додатки для Android. Подібно до Microsoft Visual Basic 6, The Simple project визначає форми (які містять компоненти) і код (який містить логіку програми). Взаємодія між компонентами та програмною логікою відбувається через події, викликані компонентами. Логіка програми складається з обробників подій, які містять код реагування на події. The Simple project не надто активний. Останнє оновлення вихідний код зазнавав у серпні 2009 року [24].

Visual Studio 2015 (рис. 2.16) підтримує розроблення крос-платформних прикладних програм, дозволяючи розробникам C++ створювати проекти із шаблонів для Android-прикладних програм або створювати високопродуктивні колективні бібліотеки для включення їх до інших рішень. Функціонал середовища містить інтелектуальний підказувач IntelliSense, точки зупину, розгортання пристроїв та емуляції [41].

WinDev Mobile – це власна IDE, створена PC SOFT (рис. 2.17) та використовується для створення графічного інтерфейсу користувача (GUI) прикладних програм для смартфонів та планшетів (включаючи пристрої Android). Вона використовує мову програмування WLanguage та доступна англійською, французькою та китайською мовами.

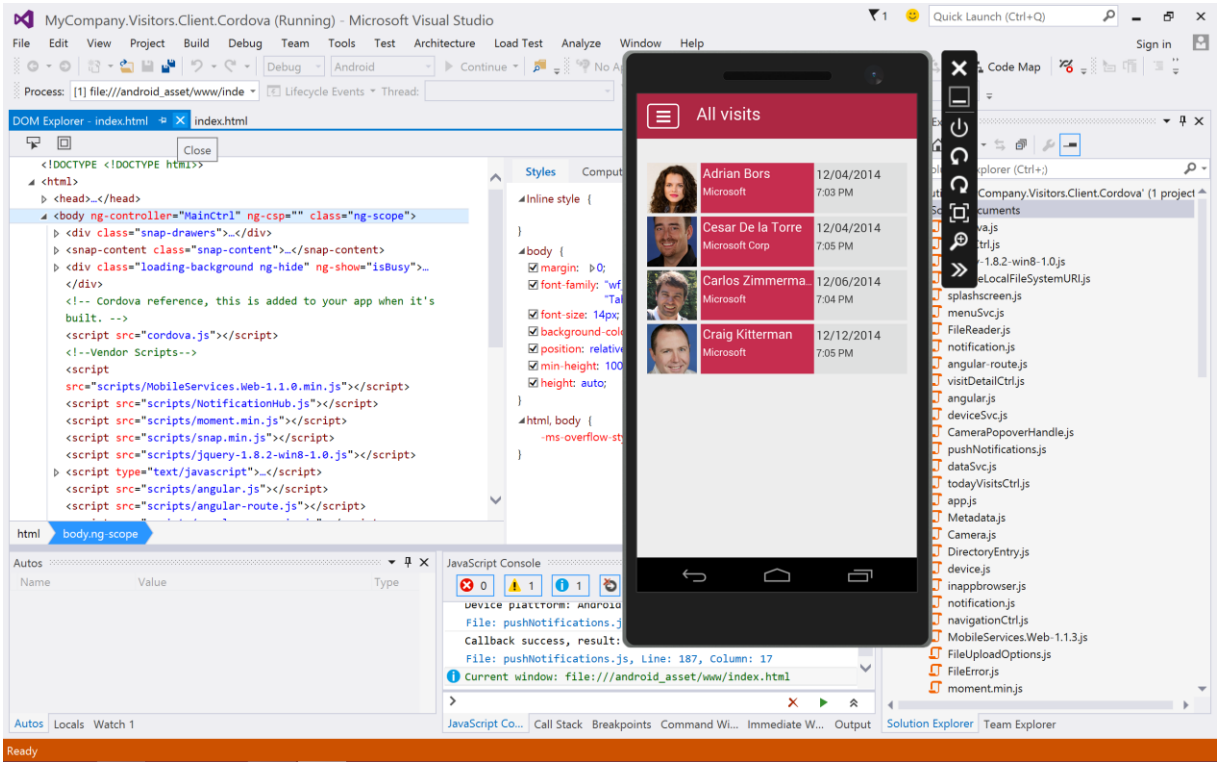

Рис. 2.16. **Visual Studio 2015**

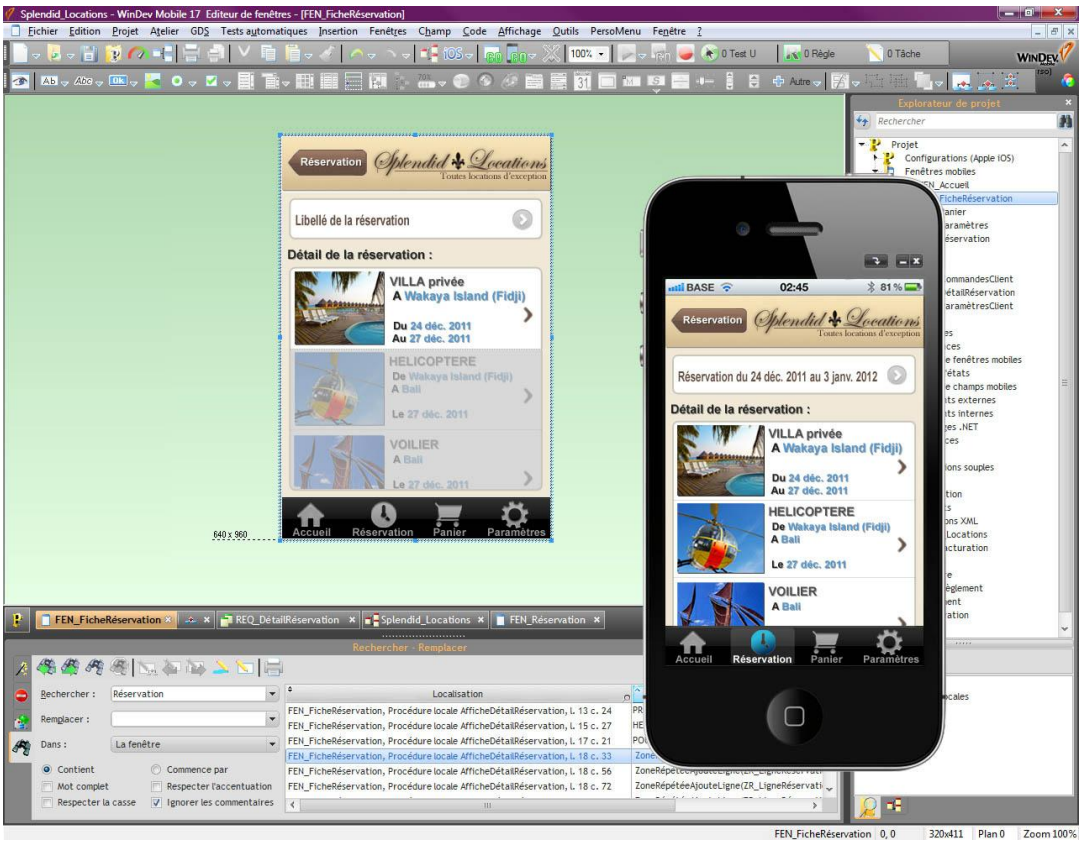

Рис. 2.17. **WinDev Mobile**

Розробники на C# можуть використовувати Xamarin для створення прикладних програм для платформ IOS, Android, Windows. Xamarin використовують більш ніж 505,000 розробників у понад 120 країнах по всьому світу станом на лютий 2014 року.

X11 Basic є діалектом мови програмування Basic із графічними можливостями, який об'єднує такі функції, як: оболонки сценаріїв, програмування CGI та повної графічної візуалізації. Синтаксис в основному схожий на старий GFA Basic, який використовувався на комп'ютерах Atari ST.

2.3. Огляд інтегрованих середовищ розроблення (IDE)

Інтегроване середовище розроблення (IDE) є інструментом, який дозволяє розробникам прикладних програм виконати повний цикл розроблення програмного забезпечення. Цей цикл охоплює проектування, кодування, компіляцію, тестування, налагодження та упакування програмного забезпечення прикладної програми. Для створення Android додатків Google в цей час підтримує дві IDE:

Android Developer Tools (ADT) http://developer.android.com/sdk /index.html;

Android Studio https://developer.android.com/studio/index.html.

Обидва ці середовища розроблення потребують використання мови Java.

Середовища розроблення від Google та мова програмування Java не є єдиними варіантами для розроблення прикладних програм Android App. Деякі розробники не знають мови програмування Java або вважають за краще програмувати з використанням мови C/C++. Деякі розробники хотіли б використовувати єдиний базовий код для підтримки інших платформ: Apple (IOS), Windows, Blackberry і Web (HTML5). Таке розроблення відоме як крос-платформне розроблення. Існує багато альтернатив інструментам Google. Огляд альтернативних IDE наведено в табл. 2.4. У таких системах код може бути написано на таких різних мовах, як BASIC, HTML5 або Lua. Багато з альтернативних IDE можуть використовувати вільно, деякі мають відкритий вихідний код, а деякі є обмеженими версіями. Деякі потребують Android Software Development Kit (SDK), який поставляє разом з інструментами Google. Можна встановити кілька середовищ розроблення на одному комп'ютері, щоб протестувати їхні можливості.

Таблиця 2.4

Огляд альтернативних середовищ розроблення

Найменування	Мова програмування	Крос-платформність	URL
1	2	3	4
AIDE (Android IDE)	HTML5/C/C++	Так	http://www.android-ide.com/
Application Craft	HTML5	Так	http://www.applicationcraft.com/
Basic4Android	BASIC	Ні	http://www.basic4ppc.com/
Cordova	HTML5	Так	https://cordova.apache.org/
Corona	Lua	Так	http://coronalabs.com/
Intel XDK	HTML5	Так	https://software.intel.com/enus/html5/tools
IntelliJIDEA	Java	Ні	https://www.jetbrains.com/idea/features/android.html
Kivy	Python	Так	http://kivy.org/#home
MIT App Inventor	Blocks	Так	http://appinventor.mit.edu/explore/
Monkey X	BASIC	Так	http://www.monkeycoder.co.nz/
MonoGame	C#	Так	http://www.monogame.net/
MoSync	HTML5/C/C++	Так	http://www.mosync.com/
NS BASIC	BASIC	Так	https://www.nsbasic.com/
PhoneGap	HTML5	Так	http://phonegap.com/
RAD Studio XE	Object Pascal, C++	Так	http://www.embarcadero.com/

Закінчення табл. 2.4

1	2	3	4
RFO Basic	BASIC	Ні	http://laughton.com/basic/
RhoMobile Suite	Ruby	Так	http://www.motorolasolutions.com/US-EN/Business+Product+and+Services/Software+and+Applications/RhoMobile+Suite
Telerik	HTML5	Так	http://www.telerik.com/platform#overview
Titanium	JavaScript	Так	http://www.appcelerator.com/titanium/titanium-sdk/
Xamarin	C#	Так	http://xamarin.com/

2.4. Завдання та стек переходів назад (Tasks and Back Stack)

Зазвичай додаток містить кілька операцій. Кожна операція (Activity) має розроблятися, у зв'язку з дією певного типу, яку користувач може виконувати та 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, наявні в інших додатках на пристрої. Наприклад, якщо додаток хоче відправити повідомлення електронної пошти, можна визначити намір для виконання дії "відправити" і включити до 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. У цьому випадку намір полягає в тому, щоб відправити повідомлення електронної пошти, 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, із якими взаємодіє користувач під час виконання певного завдання. Операції впорядковано у вигляді стека

(стека переходів назад), у тому порядку, у якому відкривалися операції [2 – 4; 8; 40].

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 та відкривається "основна" операція для цього додатка як коренева операція у стеці.

Коли поточна операція запускає іншу, нова операція поміщається у вершину стека й отримує фокус. Попередня операція залишається у стеці, але її виконання зупиняється. Коли операція зупиняється, система зберігає поточний стан її інтерфейсу користувача. Коли користувач натискає кнопку *Назад*, поточна операція видаляється з вершини стека (операція знищується) і поновлюється робота попередньої операції (відновлюється попередній стан її інтерфейсу користувача). Операції у стеці ніколи не змінюють порядок, їх тільки додають у стек і видаляють із нього; додають під час запуску поточної операції і видаляють, коли користувач виходить із неї за допомогою кнопки *Назад*. Насправді, стек переходів назад працює за принципом "останнім увійшов – першим вийшов". На рис. 2.18 цю поведінку показано на часовій шкалі: стан операцій і поточний стан стека переходів назад показано в кожен момент часу.

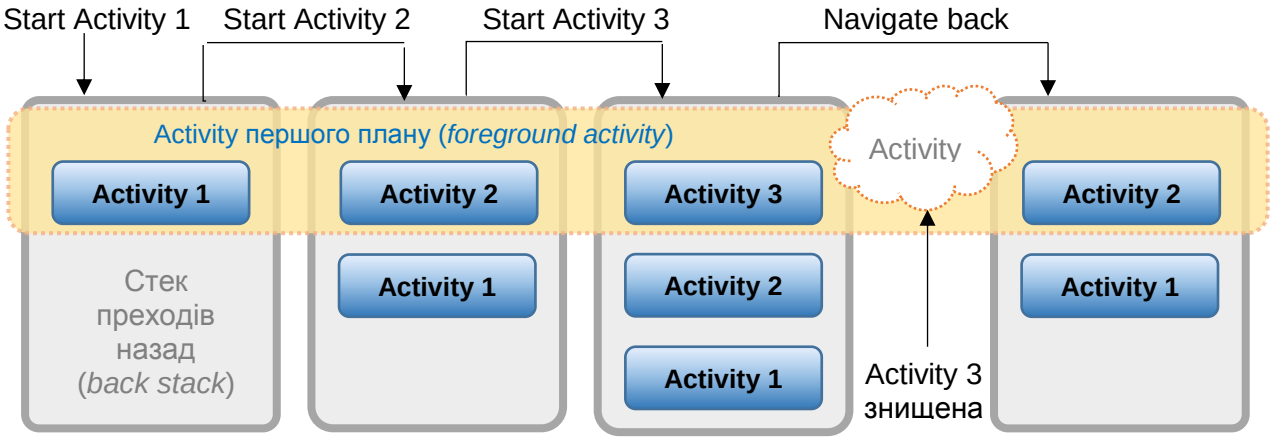

Рис. 2.18. **Поведінка стека переходів назад**

На рис. 2.19 подана ілюстрація того, 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 видалено зі стека, завдання припиняє існування.

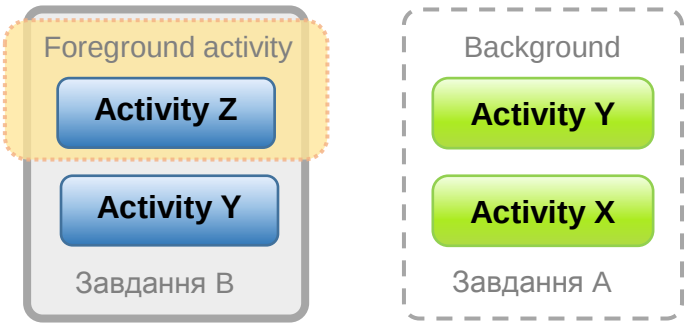

Рис. 2.19. **Робота завдання A, що перебуває у фоновому режимі, та завдання B**

На рис. 2.20 бачимо два завдання: Завдання B взаємодіє з користувачем на передньому плані, тоді як Завдання A перебуває у фоновому режимі, чекаючи відновлення.

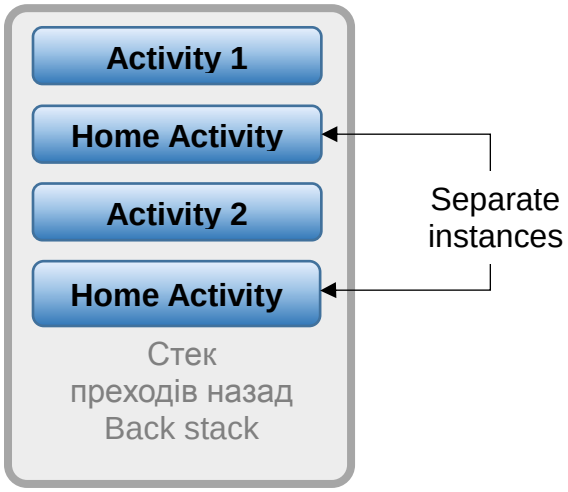

Рис. 2.20. **Створення кількох примірників однієї операції**

Завдання – це пов'язаний блок, який може переходити у фоновий режим, коли користувачі починають нове завдання або переходять на головний екран за допомогою кнопки *Home*. У фоновому режимі всі операції завдання зупинено, але стек зворотного виклику для завдання

залишається незмінним. Завдання просто втратило фокус під час виконання іншого завдання, як показано на рис 2.20. Потім завдання може повернутися на передній план, так що користувачі можуть продовжити його з перерваного місця. Слід припустити, наприклад, що поточне завдання (Завдання А) містить три операції у своєму стеку – дві операції під поточною операцією. Користувач натискає кнопку *Додому*, потім запускає новий додаток із засобу запуску додатків. Коли з'являється головний екран, Завдання А переходить у фоновий режим. Коли запускається новий додаток, система запускає завдання для цього додатка (Завдання В) зі своїм власним стеком операцій. Після взаємодії із цим додатком користувач знову повертається на головний екран і вибирає спочатку запущене Завдання А. Тепер Завдання А переходить на передній план – усі три операції її стека залишилися незмінними, і поновлюється операція, яка перебуває на вершині стека. У цей момент користувач може також переключитися назад на Завдання В, перейшовши на головний екран і вибравши значок програми, яка запустила це завдання (або вибравши завдання додатка на екрані огляду). Це приклад багатозадачності в системі Android.

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

Оскільки операції у стеці ніколи не змінюють порядок, якщо додаток дозволяє користувачам запускати певну операцію з декількох операцій, новий екземпляр такої операції створюється та поміщається у стек (замість розміщення будь-якого з попередніх примірників операції на вершину стека). Насправді, для однієї операції вашого додатка може бути створено кілька примірників (навіть із різних завдань), як показано на рис. 2.20. Тому, якщо користувач переходить назад за допомогою кнопки *Назад*, кожен екземпляр операції з'являється в тому порядку, у якому вони були відкриті (кожен зі своїм станом призначеного для інтерфейсу користувача). Однак можна змінити цю поведінку, якщо не потрібно, щоб створювалося кілька примірників операції.

Слід підбити підсумки поведінки операцій і завдань:

коли Операція А запускає Операцію В, Операція А зупиняється, але система зберігає її стан (наприклад, положення прокручування й текст, уведений у форми). Якщо користувач натискає кнопку *Назад* в Операції В, Операція А відновлює роботу зі збереженого стану;

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

якщо користувач натискає кнопку *Назад*, поточна операція видаляється зі стека та знищується. Відновлюється попередня операція у стеці. Коли операція знищується, система не зберігає стан операції;

можна створювати кілька примірників операції, навіть з інших завдань.

Дизайн навігації

Додаткову інформацію про роботу навігації отримують у додатку Android.

2.4.1. Збереження стану операції

Як говорилося раніше, система за замовчуванням зберігає стан операції, коли вона зупиняється. Таким чином, коли користувачі повертаються назад у попередню операцію, відновлюється її інтерфейс користувача у момент зупинки. Однак можна і потрібно із попередженням зберігати стан ваших операцій за допомогою методів зворотного виклику на випадок знищення операції та необхідності у її повторному створенні.

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

Додаткову інформацію про збереження стану вашої операції див. у документі Операції.

2.4.2. Управління завданнями

Для більшості додатків спосіб, яким Android управляє завданнями та стеком переходів назад, описаний раніше, – уміщення всіх операцій

послідовно в одне завдання у стек "останнім увійшов – першим вийшов", – працює добре, і не має турбуватися про зв'язок операцій із завданнями або про їхнє існування у стеці переходів назад. Однак можна вирішити, що потрібно перервати звичайну поведінку. Можливо, для того, щоб операція в додатку починала нове завдання під час запуску (замість уміщення в поточне завдання), або під час запуску операції перенести на передній план її наявний екземпляр (замість створення нового екземпляра на вершині стека переходів назад), або щоб під час виходу користувача із завдання зі стека переходів видалялися всі операції, крім кореневої операції.

Можна здійснювати ці та багато інших дій за допомогою атрибутів в елементі маніфесту `<activity>` і за допомогою прапорців у намірі, переданому у `startActivity()`.

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

`taskAffinity`;

`launchMode`;

`allowTaskReparenting`;

`clearTaskOnLaunch`;

`alwaysRetainTaskState`;

`finishOnTaskLaunch`.

А головними прапорцями намірів, які можна використовувати, є такі:

`FLAG_ACTIVITY_NEW_TASK`;

`FLAG_ACTIVITY_CLEAR_TOP`;

`FLAG_ACTIVITY_SINGLE_TOP`.

У наступних розділах показано, як можна використовувати ці атрибути маніфесту та прапорці намірів для визначення зв'язку операцій із завданнями та їхньої поведінки у стеці переходів назад.

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

Увага! У більшості додатків не слід переривати поведінку операцій і завдань за замовчуванням. Якщо виявлено, що операції необхідно змінити поведінку за замовчуванням, будьте уважні та протестуйте зручність роботи з операцією під час запуску та зворотної навігації до неї з інших операцій і завдань за допомогою кнопки *Назад*. Обов'язково про-

тестувати поведінку навігації, яка може суперечити поведінці, очікуваній користувачем.

2.4.3. Визначення режимів запуску

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

використання файла маніфесту. Коли оголошено операцію у файлі маніфесту, можна вказати, як операція має зв'язуватися із завданнями під час її запуску;

використання прапорців намірів. Коли викликано `startActivity()`, можна включити прапорець в `Intent`, який оголошує, як має бути пов'язана нова операція з поточним завданням (і чи повинна).

Насправді, якщо Операція А запускає Операцію В, Операція В може визначити у своєму маніфесті, як вона має бути пов'язана з поточним завданням (якщо взагалі повинна), а Операція А може також запросити, як Операція В має бути пов'язана з поточним завданням. Якщо обидві операції визначають, як Операція В має бути пов'язана із завданням, тоді запит Операції А (як визначено в намірі) обробляють через запит Операції В (як визначено у її маніфесті).

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

2.4.4. Використання файла маніфесту

Під час оголошення операції у файлі маніфесту можна вказати, як операція має бути пов'язана із завданням за допомогою атрибута `launchMode` елемента `<activity>`.

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

`"standard"`. Режим за замовчуванням. Система створює новий екземпляр операції в завдання, із якої вона була запущена, і направляє йому намір. Може бути створено кілька примірників операції, кожен екземпляр може належати різним завданням і одне завдання може містити кілька примірників;

"singleTop". Якщо екземпляр операції вже існує на вершині поточного завдання, система направляє намір у цей екземпляр шляхом виклику її методу onNewIntent(), а не шляхом створення нового екземпляра операції. Може бути створено кілька примірників операції, кожен екземпляр може належати різним завданням, а одне завдання містить кілька примірників (але тільки якщо операція на вершині стека переходів назад не є наявним примірником операції).

Слід припустити, що стек переходів назад завдання 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 та стек набирає вигляду A-B-C-D-D. Однак, якщо D має режим запуску "singleTop", наявний екземпляр D отримує намір через onNewIntent(), оскільки цей екземпляр знаходиться на вершині стека – стек зберігає вигляд A-B-C-D. Однак, якщо надходить намір для операції типу B, тоді у 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 наявний екземпляр операції обробляє новий намір, користувач не може натиснути кнопку *Назад* для повернення до стану операції до надходження нового наміру в onNewIntent().

"singleTask". Система створює нове завдання та створює екземпляр операції в корені нового завдання. Однак, якщо екземпляр операції вже існує в окремому завданні, система направляє намір в наявний екземпляр шляхом виклику його методу onNewIntent(), а не шляхом створення нового екземпляра. Одночасно може існувати тільки один екземпляр операції.

Примітка. Хоча операція запускає нове завдання, кнопка *Назад* повертає користувача до попередньої операції.

"singleInstance". Те ж, що і "singleTask", але система не запускає ніяких інших операцій у завдання, що містить цей екземпляр. Операція завжди є єдиним членом свого завдання; будь-які операції, запущені цією операцією, відкриваються в окремому завданні.

Як інший приклад: додаток Android Browser оголошує, що операція веб-браузера має завжди відкриватися у своїй власній меті – шляхом указівки режиму запуску singleTask в елементі <activity>. Це означає,

що якщо додаток видає намір відкрити Android Browser, його операція *не* вміщається в те ж завдання, що й додаток. Замість цього або для браузера запускається нове завдання, або, якщо браузер уже має завдання, що працює у фоновому режимі, це завдання переходить на передній план для оброблення нового наміру.

І під час запуску операції в новому завданні, і під час запуску операції в наявному завданні, кнопка *Назад* завжди повертає користувача до попередньої операції. Однак, якщо запущено операцію, яка вказує режим запуску `"singleTask"`, усе завдання переходить на передній план, якщо екземпляр цієї операції існує у фоновому завданні. У цей момент стек переходів назад уміщає всі операції із завдання, що переходить на передній план, на вершину стека. Рис. 2.21 ілюструє сценарій цього типу.

На рис. 2.21 маємо подання того, як операцію з режимом запуску `"singleTask"` додано у стек переходів назад. Якщо операція вже є частиною фонового завдання зі своїм власним стеком переходів назад, то весь стек переходів назад також переноситься вгору, на вершину поточного завдання.

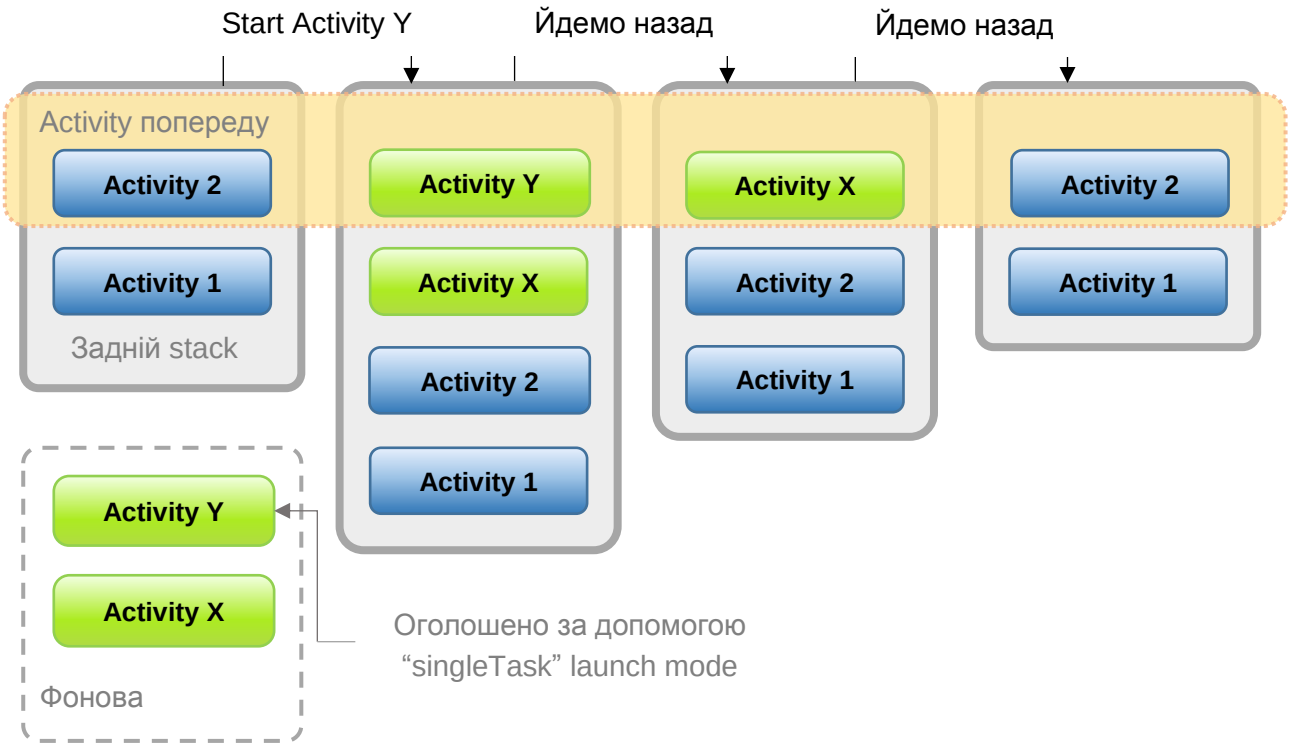

Рис. 2.21. **"SingleTask" у стеці переходів назад**

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

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

2.4.5. Використання прапорців намірів

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

`FLAG_ACTIVITY_NEW_TASK` – запуск операції в новому завданні. Якщо завдання вже працює для операції, яку запускають зараз, це завдання переходить на передній план, її останній стан відновлюється, і операція отримує новий намір в `onNewIntent()`. Це призводить до тієї ж поведінки, що й значення `launchMode` в режимі `"singleTask"`.

`FLAG_ACTIVITY_SINGLE_TOP` – якщо операція, що запускається, є поточною операцією (знаходиться на вершині стека переходів назад), тоді виклик в `onNewIntent()` отримує наявний екземпляр без створення нового екземпляра операції. Це призводить до тієї ж поведінки, що й значення `launchMode` в режимі `"singleTop"`.

`FLAG_ACTIVITY_CLEAR_TOP` – якщо операція, що запускається, вже працює в поточному завданні, тоді, замість запуску нового екземпляра цієї операції, знищуються всі інші операції, розташовані у стеці вище за неї, і цей намір доставляється у відновлений примірник 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 використовують спільно із прапорцем `FLAG_ACTIVITY_NEW_TASK`. У разі використання разом ці прапорці дозволяють знайти наявну операцію в іншому завданні та помістити її у положення, де вона зможе реагувати на намір.

Примітка. Якщо для призначеної операції встановлено режим запуску `"standard"`, вона також видаляється зі стека та на її місце запускається новий екземпляр, щоб обробити вхідний намір. Саме тому в режимі запуску `"standard"` завжди створюється новий екземпляр для нового наміру.

Оброблення прив'язувань. Прив'язування вказує на переважну належність операції до завдання. За замовчуванням усі операції з однієї програми мають прив'язування один до одного. Тому за замовчуванням усі операції однієї програми воліють перебувати в одному завданні. Однак можна змінити прив'язування за замовчуванням для операції. Операції, визначені в різних додатках, можуть спільно використовувати одне прив'язування; таким же чином операції, визначені в одному додатку, можуть мати прив'язування до різних завдань. Можна змінити прив'язування будь-якій цій операції за допомогою атрибута `taskAffinity` елемента `<activity>`.

Атрибут `taskAffinity` набирає строкового значення, яке має відрізнятися від назви пакета за замовчуванням, оголошеного в елементі `<manifest>`, оскільки система використовує цю назву для ідентифікації прив'язування завдання за замовчуванням для додатка.

Прив'язування вступає у гру у двох випадках.

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

Нова операція за замовчуванням запускається в завдання тієї операції, яка викликала `startActivity()`. Вона поміщається в той же стек переходів назад, що й операція, що її викликає. Однак, якщо намір, переданий у `startActivity()`, містить прапорець `FLAG_ACTIVITY_NEW_TASK`, система шукає інше завдання для розміщення нової операції. Частіше це нове завдання, але необов'язково. Якщо вже наявне завдання з тією ж прив'язкою, що й у новій операції, операція запускається в цьому ж завданні. Якщо немає – операція починає нове завдання.

Якщо цей прапорець призводить до того, що операція починає нове завдання і користувач натискає кнопку `Додому` для виходу з неї, має існувати спосіб, що дозволяє користувачеві повернутися до завдання. Деякі об'єкти (такі як диспетчер повідомлень) завжди запускають операції в зовнішньому завданні, а не в складі власного, тому вони завжди поміщають прапорець `FLAG_ACTIVITY_NEW_TASK` у наміри, які вони передають в `startActivity()`. Якщо є операція, яку можна викликати зовнішнім об'єктом, що використовує цей прапорець, слід подбати про те, щоб у користувача був незалежний спосіб повернутися в запущене завдання, наприклад, за допомогою значка запуску (коренева операція завдання містить фільтр намірів `CATEGORY_LAUNCHER`).

Якщо для атрибута `allowTaskReparenting` операції встановлено значення `"true"`, операція може переміститися із завдання, що запустило її, в завдання, до якого в операції є прив'язка, коли це завдання переходить на передній план.

Слід припустити, що операцію, яка повідомляє про погодні умови в обраних містах, визначено у складі додатка для мандрівників. Вона має те ж саме прив'язування, що й інші операції в тому ж додатку (прив'язування зі стандартними програмами), і допускає перепідпорядкування із цим атрибутом. Коли одна з операцій запускає операцію прогнозу погоди, вона спочатку належить тому же завданню, що й ця операція. Однак, коли завдання із програми для мандрівників переходить на передній план, операція прогнозу погоди перепризначається цьому завданню та відображається всередині нього.

Порада. Якщо файл `.apk` містить більше одного "додатка", із точки зору користувача, ймовірно, слід використовувати атрибут `taskAffinity` для призначення різних прив'язувань операціями, пов'язаними з кожним "додатком".

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

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

`alwaysRetainTaskState` – якщо для цього атрибута встановлено значення `"true"` в кореневій операції завдання, описана раніше поведінка за замовчуванням не відбувається. Завдання відновлює всі операції у своєму стеку навіть після закінчення тривалого періоду часу.

`clearTaskOnLaunch` – якщо для цього атрибута встановлено значення `"true"` в кореневій операції завдання, стек очищається до кореневої операції кожен раз, коли користувач виходить із завдання і повертається в нього. Інакше кажучи, цей атрибут протилежний атрибуту `alwaysRetainTaskState`. Користувач завжди повертається в завдання у його початковому стані, навіть після короткочасного виходу з нього.

`finishOnTaskLaunch` – цей атрибут схожий на `clearTaskOnLaunch`, але він діє на одну операцію, а не на все завдання. Він також може призводити до видалення будь-якої операції, включаючи кореневу операцію.

Коли для нього встановлено значення `"true"`, операція залишається частиною завдання тільки для поточного сеансу. Якщо користувач виходить із завдання, а потім повертається до нього, операція вже відсутня.

2.4.6. Запуск завдання

Можна зробити операцію точкою входу, призначаючи їй фільтр намірів зі значенням `"android.intent.action.MAIN"` як зазначеної дії і `"android.intent.category.LAUNCHER"` як зазначеної категорії. Наприклад (рис. 2.22):

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

Рис. 2.22. **Призначення фільтра намірів завдання**

Фільтр намірів такого типу призводить до того, що в засобі запуску програми відображаються значок та мітка для операції, що дозволяє користувачеві запускати операцію і повертатися в завдання, яке її створило, у будь-який момент після її запуску.

Ця друга можливість дуже важлива: користувачі мають мати можливість виходити із завдання і потім повертатися в нього за допомогою цього засобу запуску операції. Тому два *режима запуску*, які зазначають, що операції завжди ініціюють завдання, `"singleTask"` и `"singleInstance"`, мають використовувати 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 нове завдання, і користувач деякий час працює в цьому завданні. Потім користувач натискає кнопку *Додому*. Завдання переводиться у фоновий режим і не відображається. Тепер у користувача немає можливості повернутися до завдання, оскільки воно відсутнє в засобі запуску додатка.

Для таких випадків, коли ви не хочете, щоб користувач міг повернутися до операції, установіть для атрибута `finishOnTaskLaunch` елемента `<activity>` значення `"true"`.

Додаткова інформація про уявлення завдань і операцій та управління ними на екрані огляду.

2.4.7. Екран огляду (Overview screen)

Екран огляду (також використовують назви: екран останніх завдань, список останніх завдань або найновіші програми) є елементом інтерфейсу користувача системного рівня, у якому міститься список останніх *операцій* і *завдань*. Користувач може переміщатися по списку та вибирати завдання для відновлення, або жестом видаляти завдання зі списку. У версії Android 5.0 (рівень API 21) кілька примірників однієї операції, що містять різні документи, можуть відображатися у вигляді завдань на екрані огляду. Наприклад, *Google Диск* може мати завдання для кожного з декількох документів Google. У вікні кожен документ відображається у вигляді завдання (рис. 2.23).

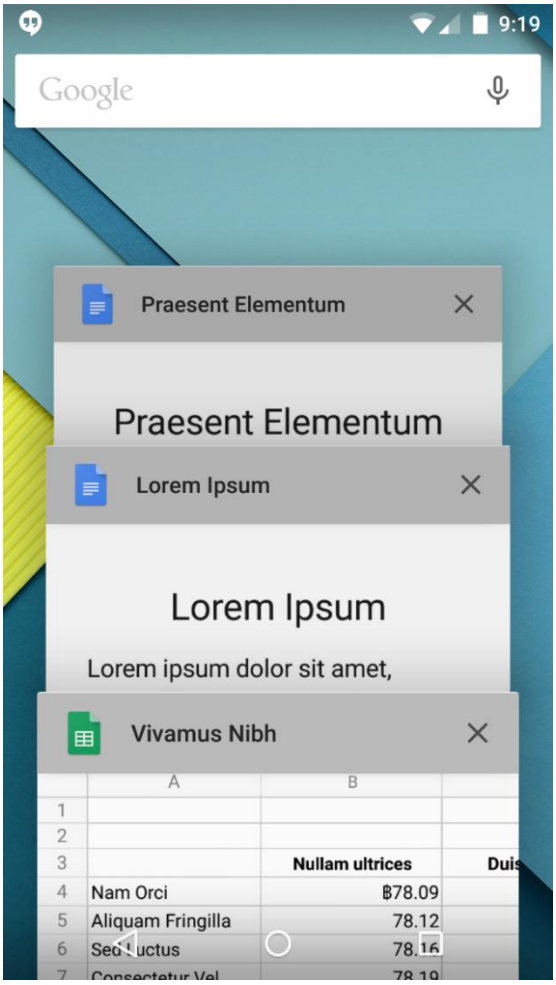

Рис. 2.23. **Екран огляду, на якому показано три документи *Google Диск*, які подані у вигляді окремих завдань**

Зазвичай, слід дозволити системі визначити спосіб уявлення завдань 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 завданнями, а за допомогою прапорців операції класу `Intent` вказується, коли операція додається на екран огляду або видаляється з нього. Крім того, атрибути `<activity>` дозволяють установлювати поведінку в маніфесті.

2.4.8. Додавання завдань на екран огляду

Використання прапорців класу `Intent` для додавання завдання 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 наявного завдання для документа.

Використання прапорця Intent для додавання завдання. Під час створення нового документа для операції слід викликати метод `startActivity()` класу `ActivityManager.AppTask`, передаючи йому intent, який запускає операцію. Для вставлення логічного розриву, щоб система обробляла цю операцію як нове завдання на вікні, потрібно передати прапорець `FLAG_ACTIVITY_NEW_DOCUMENT` у метод `addFlags()` `Intent`, який запускає операцію.

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

Якщо встановили прапорець `FLAG_ACTIVITY_MULTIPLE_TASK` під час створення нового документа, система завжди створює нове завдання із цільовою операцією як кореня. Цей параметр дозволяє відкривати один документ у декількох завданнях. Такий код показує, як це робить основна операція (рис. 2.24):

```
// DocumentCentricActivity.java
public void createNewDocument(View view) {
  final Intent newDocumentIntent = newDocumentIntent();
  if (useMultipleTasks) {
    newDocumentIntent.addFlags(Intent.FLAG_ACTIVITY_MULTIPLE_TASK);
```

Рис. 2.24. **Створення нового завдання із цільовою операцією як кореня**

```
  }
  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++;
}
```

Закінчення рис. 2.24

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

Коли основна операція запускає нову операцію, система шукає в наявних завданнях одну, значення `intent` якої відповідає назві компонента і даними `Intent` для операції. Якщо завдання не знайдено або `intent` містить прапорець `FLAG_ACTIVITY_MULTIPLE_TASK`, створюється нове завдання з операцією як кореня. Якщо завдання знайдено, система виводить це завдання на передній план і передає нове значення `intent` в `onNewIntent()`. Нова операція отримує `intent` і створює новий документ на екрані огляду, як у прикладі на рис. 2.25:

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

Рис. 2.25. **Створення нового документа на екрані огляду**

```
            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);

  /* Якщо прапорець FLAG_ACTIVITY_MULTIPLE_TASK не буде встановлений, то
ця activity буде повторно створювати новий документ. */
    setDocumentCounterText(R.string.reusing_document_counter);
}
```

Закінчення рис. 2.25

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

`"intoExisting"` – операція повторно використовує наявне завдання для документа. Це тотожно установленю прапорця `FLAG_ACTIVITY_NEW_DOCUMENT` *без* установлення *прапорця* `FLAG_ACTIVITY_MULTIPLE_TASK`.

`"always"` – операція створює нове завдання для документа, навіть якщо документ вже відкрито. Використання цього значення тотожно установленню обох прапорців `FLAG_ACTIVITY_NEW_DOCUMENT` й `FLAG_ACTIVITY_MULTIPLE_TASK`.

`"none"` – операція не створює нове завдання для документа. Екран огляду обробляє операцію як операцію за замовчуванням: на екрані огляду відображається одне завдання для програми, яка відновлюється з будь-якої останньої операції, викликаної користувачем.

`"never"` – операція не створює нове завдання для документа. Установлення цього значення визначає поведінку прапорців `FLAG_ACTIVITY_NEW_DOCUMENT` й `FLAG_ACTIVITY_MULTIPLE_TASK`, якщо обидва вони встановлені в *intent* і на екрані огляду відображається одне завдання для програми, яка відновлюється з будь-якої останньої операції, викликаної користувачем.

Примітка. Для значень, крім `none` та `never`, операцію має бути визначено з атрибутом `launchMode="standard"`. Якщо цей атрибут не вказано, використовується `documentLaunchMode="none"`.

2.4.9. Видалення завдань

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

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

Можна встановити максимальну кількість завдань, яку додаток може включити в екран огляду, установивши для атрибута `<activity> android:maxRecents` ціле значення. Значення за замовчуванням: 16. За досягнення максимальної кількості завдань найбільш довго не використовуване завдання видаляється з екрана огляду. Максимальне значення `android:maxRecents` становить 50 (25 для пристроїв із малим обсягом пам'яті). Значення менші за 1 не допускають.

Використання класу AppTask для видалення завдань. В операції, яка створює нове завдання на вікні, можна вказати час видалення завдання та завершення всіх пов'язаних із нею операцій, викликавши метод `finishAndRemoveTask()` (рис. 2.26).

```
// NewDocumentActivity.java

public void onRemoveFromRecents(View view) {
    // Документ больше не нужен; удалить задачу.
    finishAndRemoveTask();
}
```

Рис. 2.26. **Зазначення часу видалення завдання й завершення всіх пов'язаних із ним операцій**

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

Збереження завершених завдань. Щоб зберегти завдання на вікні, навіть якщо операцію завершено, передайте прапорець `FLAG_ ACTIVITY_`

RETAIN_IN_RECENTS у метод addFlags() об'єкта *Intent*, який запускає операцію (рис. 2.27).

```
// DocumentCentricActivity.java

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;
}
```

Рис. 2.27. **Збереження завдання на вікні**

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

2.5. Компоненти інтерфейсу

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

Оголошення елементів інтерфейсу користувача у XML. В Android є зручний довідник XML-елементів для класів View і їхніх підкласів, наприклад таких, які використовують для віджетів і макетів.

Створення екземплярів елементів під час виконання. Програма може програмним чином створювати об'єкти View і ViewGroup (а також управляти їхніми властивостями).

Платформа Android надає гнучкість під час використання будь-якого із цих способів для оголошення інтерфейсу користувача додатка і його управління. Наприклад, можна оголосити у XML макети за замовчуванням, включаючи елементи екрана, які будуть відображатися

в макетах, і їхні властивості. Потім можна додати в додаток код, який дозволяє змінювати стан об'єктів на екрані (включаючи оголошені у XML) під час виконання.

В *підключеному модулі ADT для Eclipse* передбачено функцію попереднього перегляду створеного файлу XML: досить відкрити файл XML і вибрати вкладку *Layout* (Макет).

Для налаштовування макетів можна скористатися інструментом *Hierarchy Viewer* і з його допомогою переглянути значення властивостей, рамки з індикаторами заповнення або полів, а також повністю зображені уявлення прямо під час налагодження програми в емуляторі або на пристрої.

За допомогою інструменту *layoutopt* можна швидко проаналізувати макети та їхні ієрархії щодо низької ефективності чи інших проблем.

Перевага оголошення інтерфейсу користувача у файлі XML полягає в тому, що таким чином можна ефективно відокремити уявлення свого додатка від коду, який управляє його поведінкою. Описи інтерфейсу користувача, перебувають 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 у XML спрощує візуалізацію структури інтерфейсу користувача, завдяки чому вирішення проблем також стає простішим. У даній статті ми навчимо вас оголошувати макет у XML. Якщо віддаєте перевагу створювати екземпляри об'єктів View під час виконання, зверніться до довідкової документації для класів `ViewGroup` і `View`.

Переважно, довідник XML-елементів для оголошення елементів інтерфейсу користувача точно дотримується структури та правил іменування для класів і методів (назви елементів відповідають назвам 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, в елемента `EditText` є атрибут `text`, який відповідає методу `EditText.setText()`.

2.5.1. Створення XML

За допомогою довідника XML-елементів, який є в Android, можна швидко та просто створювати макети для інтерфейсу користувача і елементи, які містяться в ньому, точно так само, як під час створення веб-сторінок у HTML, – за допомогою вкладених елементів.

У кожному файлі макета має бути всього один кореневий елемент – об'єкт уявлення (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, що визначає макет. Далі наведено приклад макета XML, у якому використано вертикальний об'єкт `LinearLayout`, де розміщено елементи `TextView` і `Button` (рис. 2.28).

```
<?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>
```

Рис. 2.28. **Вертикальний об'єкт `LinearLayout` разом з `TextView` і `Button`**

Після оголошення макета у файлі XML слід зберегти файл із розширенням `.xml` у каталог `res/layout/` проекту Android для подальшої компіляції.

2.5.2. Завантаження ресурсу XML

Під час компіляції додатка кожен файл XML-макета компілюється в ресурс `View`. Необхідно завантажити ресурс макета в коді програми в ході реалізації методу зворотного виклику `Activity.onCreate()`. Для цього викликати метод `setContentView(android.view.View)`, передати в нього посилання на ресурс макета в такій формі: `R.layout.layout_file_name`.

Наприклад, якщо макет XML збережено як файл `main_layout.xml`, завантажити його для операції необхідно таким чином (рис. 2.29):

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

Рис. 2.29. **Завантаження операцій `main_layout.xml`**

Метод зворотного виклику `onCreate()` в операції викликається платформою Android під час запуску операції.

2.5.3. Атрибути

Кожен об'єкт View і ViewGroup підтримують 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 розглядають як "параметри макета". Такі атрибути описують певні орієнтації макета для об'єкта `View`, задані батьківським об'єктом `ViewGroup` такого об'єкта.

2.5.4. Ідентифікатор

У будь-якого об'єкта `View` може бути пов'язаний із ним цілочисельний ідентифікатор, який слугує для позначення унікальності об'єкта `View` в ієрархії. Під час компіляції додатка цей ідентифікатор використовують як ціле число, однак ідентифікатор зазвичай призначається у файлі XML макета у вигляді рядка в атрибуті `id`. Цей атрибут XML є загальним для всіх об'єктів `View` (певних класом `View`), який використовують досить часто.

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

Рис. 2.30. **Синтаксис для ідентифікатора всередині тега XML**

Символ @ на початку рядка вказує на те, що оброблювачу XML слід виконати синтаксичний аналіз решти ідентифікатора і визначити її як

ресурс ідентифікатора. Символ плюса (+) означає, що це назва нового ресурсу, який необхідно створити й додати до наших ресурсів (у файлі `R.java`). В Android є ряд інших ресурсів ідентифікатора. У ході посилання на ідентифікатор ресурсу Android не потрібно вказувати символ плюса, проте необхідно додати простір назв пакета `android`, як зазначено далі на рис. 2.31:

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

Рис. 2.31. **Простір назв пакета `Android`**

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

Щоб створити уявлення та послатися на них із програми, зазвичай, слід виконати такі дії:

1. Визначити уявлення або віджет у файлі макета та надати йому унікальний ідентифікатор (рис. 2.32):

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

Рис. 2.32. **Надання унікального ідентифікатора**

2. Потім створити екземпляр об'єкта уявлення та виконати його захоплення з макета (зазвичай за допомогою методу `onCreate()` (рис. 2.33).

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

Рис. 2.33. **Створення екземпляра об'єкта уявлення та виконання його захоплення з макета**

3. Визначення ідентифікаторів для об'єктів уявлення має важливе значення під час створення об'єкта `RelativeLayout`. У відносному макеті універсальні ідентифікатори використовуються для розташування уявлень один про одного.

4. Ідентифікатор не обов'язково має бути унікальним у межах усієї ієрархії, а тільки в тій її частині, де виконують пошук (найчастіше це може бути як раз уся ієрархія, тому по можливості ідентифікатори мають бути повністю унікальними).

2.5.5. Параметри макета

Атрибути макета XML, які називають `layout_something`, визначають параметри макета для об'єкта уявлення, які підходять для класу ViewGroup, у якому він перебуває.

Кожен клас ViewGroup реалізує вкладений клас, який успадковує `ViewGroup.LayoutParams`. У цьому підкласі є типи властивостей, які визначають розмір і положення кожного дочірнього уявлення, що підходять для його групи. На рис. 2.34 показано, що батьківська група уявлень визначає параметри макета для кожного дочірнього уявлення (включаючи дочірню групу уявлень).

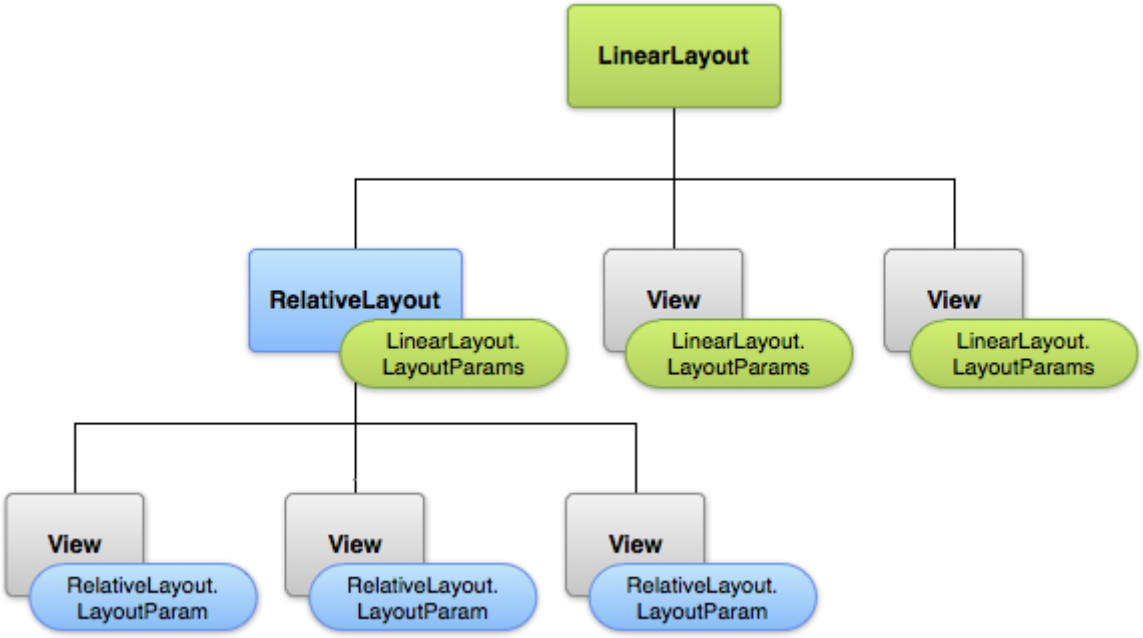

Рис. 2.34. **Графічне подання ієрархії уявлення з параметрами макета кожної вистави**

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

Усі групи уявлень містять параметри ширини та висоти (`layout_width` і `layout_height`), і кожне подання має визначати їх. Багато LayoutParams також включають додаткові параметри полів і кордонів.

Для параметрів ширини та висоти можна вказати точні значення, хоча, можливо, не слід робити це часто. Зазвичай для завдання значень ширини та висоти використовують одну з таких констант:

`wrap_content` – розмір уявлення задано за розмірами його вмісту;

`match_parent` (яка до API рівня 8 *називалася* `fill_parent`) – розмір подання визначено обмеженнями, що задаються його батьківської групою уявлень.

Переважно, не рекомендовано ставити абсолютні значення ширини та висоти макета (наприклад, у точках). Замість цього слід використовувати такі відносні одиниці вимірювання, як пікселі, які не залежать від дозволу екрана (dp), `wrap_content` або `match_parent`. Це гарантує однакове відображення додатка на пристроях з екранами різних розмірів.

2.5.6. Розміщення макета

Подання має прямокутну форму. Розташування подання визначено його координатами *зліва* та *зверху,* а його розміри – параметрами ширини та висоти. Розташування вимірюють у пікселях.

Розташування уявлення можна визначити шляхом виклику методів `getLeft()` і `getTop()`. Перший повертає координату зліва (по осі X) для прямокутника уявлення. Другий повертає верхню координату (по осі Y) для прямокутника уявлення. 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 та нижнього країв прямокутника уявлення. Наприклад, виклик методу `getRight()` аналогічне такому обчисленню: `getLeft()+getWidth()`.

Розмір, відступ і поля. Розмір уявлення визначається його шириною та висотою. Фактично, уявлення володіє двома парами значень "ширина – висота".

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

Друга пара значень – це просто *ширина* та *висота* (іноді їх називають *креслярська ширина* та *креслярська висота*). Ці розміри визначають фактичний розмір уявлення на екрані після розмічування під час їхнього відтворення. Ці значення можуть відрізнятися від виміряних ширини та висоти, хоча це й не обов'язково. Значення ширини та висоти можна визначити, викликавши методи `getWidth()` і `getHeight()`.

Під час вимірювання своїх розмірів уявлення враховує заповнення. Відступ вираженого в пікселях для лівої, верхньої, правої та нижньої частин уявлення. Відступ можна використовувати для зсуву вмісту уявлення на певну кількість пікселів. Наприклад, значення відступу зліва, що дорівнює 2, призведе до того, що вміст уявлення буде зміщено на 2 пікселі вправо від лівого краю уявлення. Для задавання відступів можна використовувати метод `setPadding(int, int, int, int)`. Щоб запросити відступ, використовують методи `getPaddingLeft()`, `getPaddingTop()`, `getPaddingRight()` і `getPaddingBottom()`.

Навіть якщо уявлення може визначити відступ, у ньому відсутня підтримка полів. Така можливість є у групи уявлень.

2.5.7. Макети інтерфейсу користувача

Лінійний макет

`LinearLayout` – це уявлення групи, яке вирівнює всі дочірні елементи в одному напряму, вертикально або горизонтально. Можна вказати напрям макета за допомогою атрибута `android:orientation` (рис. 2.35):

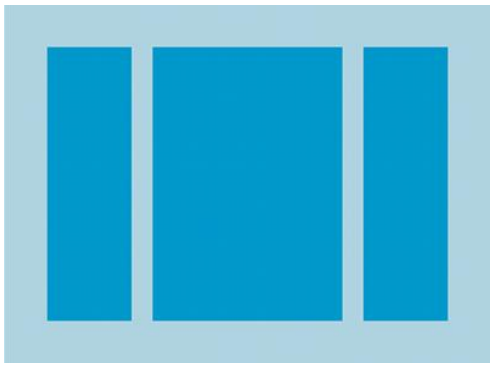

Рис. 2.35. **Лінійний макет**

Усі дочірні елементи `LinearLayout` розміщуються один за одним таким чином, що вертикальний список буде мати лише один дочірній елемент на кожен рядок, незалежно від його ширини, а горизонтальний

список матиме одну висоту (висота найвищого дочірнього елемента плюс внутрішній відступ). `LinearLayout` розуміє зовнішні відступи між дочірніми елементами та 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" уявлення з точки зору того, скільки місця воно має займати на екрані. Більше значення ваги дозволяє йому розширюватися, щоб заповнити все місце, що залишилося в батьківському уявленні. Дочірні уявлення можуть точно вказувати значення ваги, і тоді все вільне місце в уявленні групи призначено дочірнім елементам пропорційно до їхньої вказаної ваги. Значення ваги за замовчуванням дорівнює нулю.

Наприклад, якщо є три текстових поля та два з них мають вагу, що дорівнює 1, тоді як вагу інших полів не вказано, третє текстове поле без ваги не збільшиться, а займе лише область, необхідну для його вмісту. Інші два розширяться однаковою мірою, 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, тому воно отримує половину загального залишку місця, в той час як перші два розподіляють решту порівну.

Рівноважні дочірні елементи. Для того щоб створити лінійний макет, у якому кожен дочірній елемент займає однаковий об'єм простору на екрані, слід установити `android: layout_height` кожного уявлення, що дорівнює `0dp` (для вертикального макета) або `android: layout_width` кожного уявлення – `0dp` (для горизонтального макета). Потім установити `android: layout_weight` кожного уявлення, що дорівнює `1` (рис. 2.36; 2.37).

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" >
```

Рис. 2.36. **Лінійний макет, де кожний дочірний елемент займає однаковий об'єм простору на екрані**

```
    <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>
```

Закінчення рис. 2.36

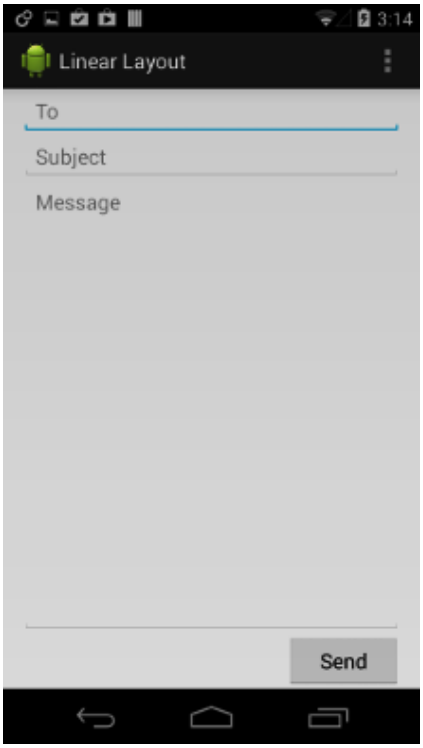

Рис. 2.37. **Лінійний макет**

Для більш детальної інформації про атрибути, доступні кожному уявленню `LinearLayout`, дивіться *LinearLayout.LayoutParams*.

Відносний макет

`RelativeLayout` – це уявлення групи, яке відображає дочірні уявлення у відносних позиціях. Положення кожного уявлення може бути визначено щодо споріднених елементів (наприклад, зліва від іншого уявлення або нижче нього) або положення щодо батьківського елемента області `RelativeLayout` (наприклад, вирівнювання по нижньому краю, зліва або по центру) (рис. 2.38).

Рис. 2.38. **Відносний макет**

`RelativeLayout` – це дуже потужна утиліта для проектування інтерфейсу користувача, оскільки вона може усунути вкладені уявлення груп і зберігати плоску ієрархію макета, що підвищує продуктивність. Якщо використовувати кілька вкладених груп `LinearLayout`, можна замінити їх одним `RelativeLayout`.

Позиціонування уявлень. `RelativeLayout` дозволяє дочірнім уявленням визначати їхнє положення щодо батьківського уявлення або відносно один одного (задається за допомогою ID). Таким чином, можна вирівняти два елементи по правому краю або розташувати один елемент нижче від іншого, розташованого в центрі екрана, по центру зліва тощо. За замовчуванням усі дочірні уявлення відмальовуються у верхньому лівому кутку макета, тому мають вказати положення кожного уявлення, використовуючи різні властивості макета, доступні з `RelativeLayout.LayoutParams`.

Деякі з безлічі властивостей макета, доступних уявлень `RelativeLayout`, містять у собі:

`android:layout_alignParentTop` – якщо значення дорівнює `true`, розміщує верхню межу цього уявлення, відповідно до верхньої межі батьківського елемента;

`android:layout_centerVertical` – якщо значення дорівнює `true`, вирівнює цей дочірній елемент вертикально по центру всередині його батьківського елемента;

`android:layout_below` – позиціонує верхній край цього подання нижче від уявлення , певного ID ресурсу;

`android:layout_toRightOf` – позиціонує лівий край цього уявлення, відповідно до правого краю нижче від уявлення, певного ID ресурсу.

Значення кожної властивості макета становить або логічний тип, щоб включати позиціонування макета щодо батьківського `RelativeLayout`, або ID, що посилається на інше уявлення в макеті, щодо якого це уявлення має бути розташоване.

У макеті XML залежності щодо інших уявлень можуть бути оголошені в будь-якому порядку. Наприклад, можна оголосити, що *view1* буде розташовано нижче від *view2*, навіть якщо уявлення *view2* оголошено останнім в ієрархії. Далі приклад демонструє подібний сценарій (рис. 2.39).

Приклад. Кожен з атрибутів, які контролюють відносне положення кожного уявлення, підкреслено.

```
<?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
```

Рис. 2.39. **Сценарій оголошення залежності щодо інших уявлень в будь-якому порядку**

```
        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>
```

Закінчення рис. 2.39

Для більш детальної інформації про атрибути, доступних кожному уявленню `RelativeLayout`, слід подивитися `RelativeLayout.LayoutParams` (рис. 2.40).

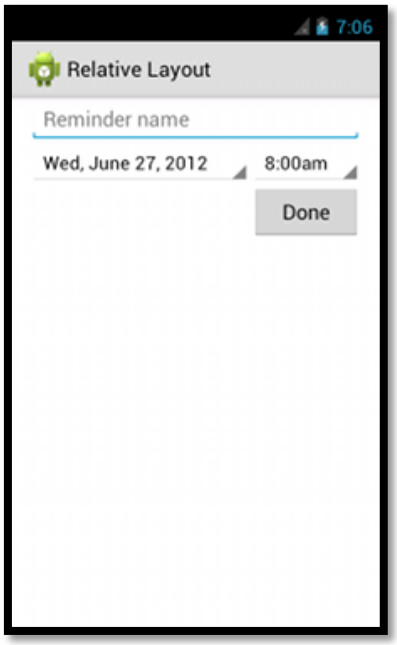

Рис. 2.40. **Відносний макет**

Подання у вигляді списку

`ListView` – це уявлення групи, яке відображає список прокручуваних елементів. Елементи списку автоматично додаються до списку за допомогою `Adapter` який витягує вміст із такого джерела, як масив або запит бази даних, і конвертує кожен елемент у уявлення, яке поміщене у список.

Для знайомства з тим, як можна динамічно додавати уявлення, використовуючи адаптер, слід прочитати `Building Layouts` *with an Adapter* (рис. 2.41):

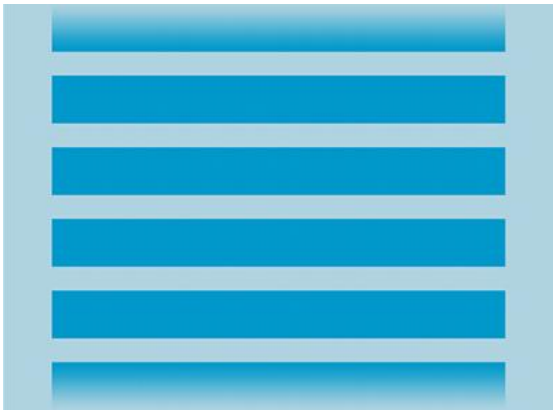

Рис. 2.41. **Подання у вигляді списку**

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

Хоча API-інтерфейси `CursorLoader` було вперше подано в Android 3.0 (API 11 рівня), вони також доступні у `Support Library`, тому що додаток може використовувати їх поки пристрої, що їх підтримують, працюють на Android версії 1.6 і вищих.

Для більш детальної інформації про використання `Loader` для асинхронного завантаження даних, слід дивитися посібник із `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`, із метою використання `CursorLoader`, який динамічно завантажує дані для уявлення у вигляді списку (рис. 2.42).

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

  // Цей адаптер, який використовується для відображення даних списку
  SimpleCursorAdapter mAdapter;

  // Це рядки контактів, які ми будемо витягувати
  static final String[] PROJECTION = new String[]{
          ContactsContract.Data._ID,
          ContactsContract.Data.DISPLAY_NAME};

  // Це критерії вибору
  static final String SELECTION = "((" +
          ContactsContract.Data.DISPLAY_NAME + " NOTNULL) AND (" +
          ContactsContract.Data.DISPLAY_NAME + " != '' ))";

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

    // Створіть індикатор виконання, поки відображається список
    ProgressBar progressBar = new ProgressBar(this);
    progressBar.setLayoutParams(new
            LayoutParams(LayoutParams.WRAP_CONTENT,
            LayoutParams.WRAP_CONTENT, Gravity.CENTER));
    progressBar.setIndeterminate(true);
    getListView().setEmptyView(progressBar);

    // Необхідно додати індикатор виконання в корінь макета
    ViewGroup root = (ViewGroup) findViewById(android.R.id.content);
    root.addView(progressBar);

    // Для адаптера курсора вкажіть, які стовпці потрапляють в якісь види
    String[] fromColumns = {ContactsContract.Data.DISPLAY_NAME};
```

Рис. 2.42. **`ListActivity`, активність та `ListView`, єдиний елемент макета за замовчуванням**

```
    // TextView в simple_list_item_1
    int[] toViews = {android.R.id.text1};

    // Створіть порожній адаптер, який ми будемо використовувати
для відображення завантажених даних
    // Ми передаємо null для курсора, після чого оновлюємо його
в onLoadFinished ()
    mAdapter = new SimpleCursorAdapter(this,
            android.R.layout.simple_list_item_1, null,
            fromColumns, toViews, 0);
    setListAdapter(mAdapter);

    // Підготуйте навантажувач. Або повторно підключіть до існуючого,
    // або створіть новий.
    getLoaderManager().initLoader(0, null, this);}

    // Викликається, коли необхідно створити новий завантажувач
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    // Тепер створіть і поверніть CursorLoader, який подбає про
    // створення курсора для відображуваних даних.
    return new CursorLoader(this, ContactsContract.Data.CONTENT_URI,
            PROJECTION, SELECTION, null, null);}

    // Викликається, коли раніше створений завантажувач завершив
завантаження
    public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
      // Змініть новий курсор. (Framework подбає про
      // закриття старого курсора, як тільки ми повернемося.)
    mAdapter.swapCursor(data);}

    // Викликається, коли раніше створений завантажувач скидається,
роблячи дані недоступними
    public void onLoaderReset(Loader<Cursor> loader) {
    // Це викликається, коли останній курсор, наданий в onLoadFinished()
    // вище, буде закрито. Нам потрібно переконатися, що ми більше не
використовуємо його.
    mAdapter.swapCursor(null);
    }
```

Продовження рис. 2.42

```
    @Override
    public void onListItemClick(ListView l, View v,
                                int position, long id) {
      // Зробіть щось, коли клацнете по елементу списку
    }
}
```

Закінчення рис. 2.42

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

Уявлення у вигляді сітки

`GridView` – це `ViewGroup`, що відображає елементи у двовимірній сітці, що перегортується. Елементи сітки автоматично додаються до макета за допомогою `ListAdapter`.

Для знайомства з тим, як динамічно додавати уявлення, використовуючи адаптер, слід читати `Building Layouts` *with an Adapter* (рис. 2.43).

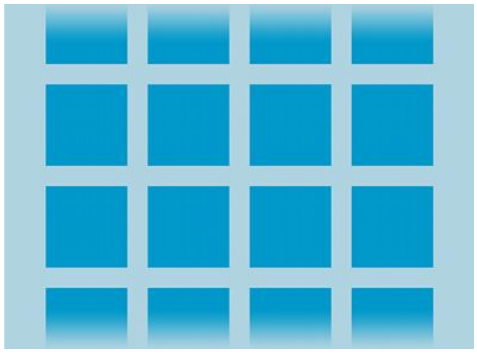

Рис. 2.43. **Уявлення у вигляді сітки**

Приклад. У цьому уроці слід створити сітку ескізів зображень. Коли елемент обрано, випливне повідомлення покаже положення елемента.

1. Створити новий проект із назвою *HelloGridView*.

2. Знайти декілька фото, які б хотіли використовувати, або *завантажте ці зразки зображень*. Зберегти файли зображень у папку проекту `res/drawable/`.

3. Відкрити файл `res/layout/main.xml` і вставити таке (рис. 2.44):

```
<?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"
/>
```

Рис. 2.44. **Вигляд кода файла** `res/layout/main.xml`

Цей `GridView` заповнить весь екран. Атрибути не потребують пояснень. Для більш детальної інформації про допустимі атрибути, дивіться посилання `GridView`.

4. Відкрити `HelloGridView.java` і вставити такий код для методу `onCreate()` (рис. 2.45):

```
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();
        }
    });
}
```

Рис. 2.45. **Вигляд кода файлів** `HelloGridView.java`

Після того як макет `main.xml` установлено для уявлення вмісту, `GridView` беруть із макета за допомогою `findViewById(int)`. Метод `setAdapter()`

потім установлює адаптер користувача (`ImageAdapter`) як джерело для всіх елементів, що відображаються в 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 зворотній виклик методу `onItemClick()`, щоб показати `Toast`, який відображає початкову позицію обраного елемента (у реальному сценарії, положення можна використовувати для отримання повнорозмірних зображень для якогось іншого завдання).

5. Створити новий клас під назвою `ImageAdapter`, який успадковує `BaseAdapter` (рис. 2.46):

```
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);
```

Рис. 2.46. **Код класу ImageAdapter, який успадковує BaseAdapter**

```
        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
    };
}
```

Закінчення рис. 2.46

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

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

`setLayoutParams(ViewGroup.LayoutParams)` установлює висоту та ширину View – це гарантує, що, незалежно від розміру області малювання, кожне зображення масштабується й обрізається до відповідних розмірів, у разі потреби;

`setScaleType(ImageView.ScaleType)` оголошує, що зображення мають бути обрізані в напряму до центру (за потреби);

`setPadding(int, int, int, int)` визначає внутрішній відступ з усіх боків (зауважте, 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 в метод, використовується для вибору зображення з масиву `mThumbIds`, який установлює ресурс зображення для `ImageView`.

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

6. Запустити додаток.

Слід пробувати експериментувати з поведінками елементів `GridView` і `ImageView`, коригуючи їхні властивості. Наприклад, замість використання `setLayoutParams(ViewGroup.LayoutParams)`, спробувати використання `setAdjustViewBounds(boolean)`.

2.6. Життєвий цикл візуальних компонентів

2.6.1. Операції (Activity)

`Activity` – це компонент програми, який видає екран, і з яким користувачі можуть взаємодіяти для виконання будь-яких дій, наприклад набрати номер телефону, зробити фото, відправити лист або переглянути карту. Кожній операції присвоєно вікно для промальовування відповідного інтерфейсу користувача. Зазвичай вікно відображається на весь екран, проте його розмір може бути меншим, і воно може розміщуватися над іншими вікнами.

Переважно, програма містить кілька операцій, які слабо пов'язані одна з одною. Зазвичай одну з операцій в додатку позначають як *основну*, пропоновану користувачеві під час першого запуску програми. У свою

чергу, кожна операція може запустити іншу операцію для виконання різних дій. Кожен раз, коли запускається нова операція, попередня зупиняється, однак система зберігає її в стек ("стек переходів назад"). Під час запуску нової операції вона поміщається у 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 та 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 використовують методи зворотного виклику життєвого циклу операції. Існує кілька таких методів, які може приймати операція, унаслідок 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, у разі зупинки операція має звільнити будь-які великі об'єкти, наприклад, підключення до мережі або бази даних. Під час відновлення операції можна повторно отримати необхідні ресурси та відновити виконання перерваних дій. Такі зміни стану є частиною життєвого циклу операції.

Далі в цьому посібнику (підрозділі) розглядають основи створення та використання операцій, включаючи повний опис життєвого циклу операції, щоб можна було краще зрозуміти, як слід управляти переходами між різними станами операції.

Створення операції

Щоб створити операцію, спочатку необхідно створити підклас класу `Activity` (або скористатися наявним його підкласом). У такому підкласі необхідно реалізувати методи зворотного виклику, які викликає система під час переходу операції з одного стану свого життєвого циклу до іншого, наприклад під час створення, зупинки, відновлення або знищення операції. Ось два найбільш важливих методи зворотного виклику:

`onCreate()` – цей метод необхідно обов'язково реалізувати, оскільки система викликає його під час створення операції. У своїй реалізації необхідно ініціалізувати ключові компоненти операції. Найбільш важливо саме тут викликати `setContentView()` для визначення макета, призначеного для інтерфейсу користувача операції;

`onPause()` – система викликає цей метод як першу ознаку виходу користувача з операції (однак це не завжди означає, що операцію буде

знищено). Зазвичай саме тут необхідно застосовувати будь-які зміни, які мають бути збережені, крім поточного сеансу роботи користувача (оскільки користувач може не повернутися назад).

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

Реалізація інтерфейсу користувача. Для реалізації призначеної для інтерфейсу користувача операції використовують ієрархію уявлень-об'єктів, отриманих із класу `View`. Кожне уявлення відповідає за певну прямокутну область вікна операції та може реагувати на дії користувачів. Наприклад, уявленням може бути кнопка, натискання на яку приводить до виконання певної дії.

В Android передбачено набір уже готових уявлень, які можна використовувати для створення дизайну макета та його організації. Віджети – це уявлення з візуальними (та інтерактивними) елементами, наприклад, кнопками, текстовими полями, чекбоксами або просто зображеннями. Макети – це уявлення, отримані з класу `ViewGroup`, що забезпечують унікальну модель компонування для своїх дочірніх уявлень, таких як лінійний макет, сітка або відносний макет. Також можна створити підклас для класів `View` та `ViewGroup` (або скористатися наявними підкласами), щоб створити власні віджети та макети і потім застосувати їх до макета своєї операції.

Найчастіше для завдання макета за допомогою уявлень використовують XML-файл макета, збережений у ресурсах додатка. Таким чином можна зберігати дизайн інтерфейсу користувача окремо від вихідного коду, який слугує для завдання поведінки операції. Щоб задати макет інтерфейсу користувача операції, можна використовувати метод `setContentView()`, передавши в нього ідентифікатор ресурсу для макета. Однак також можна створити нові `View` у коді операції та створити ієрархію уявлень. Для цього слід вставити `View` у `ViewGroup`, а потім використовувати цей макет, передавши кореневий об'єкт `ViewGroup` у метод `setContentView()`.

Оголошення операції в маніфесті. Щоб операція стала доступною системі, її необхідно оголосити у файлі маніфесту. Для цього слід відкрити файл маніфесту та додати елемент `<activity>` до дочірньої для елемента `<application>`. Наприклад (рис. 2.47):

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

Рис. 2.47. **Оголошення операції в маніфесті**

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

Додаткові відомості про оголошення операції в маніфесті див. в довідці з елемента `<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.

Під час створення нової програми за допомогою інструментів Android SDK у заготовці операції, створюваної автоматично, є фільтр намірів, який оголошує операцію. Ця операція реагує на виконання *основної* дії, і її слід помістити в категорію переходу засобу запуску. Фільтр намірів має такий вигляд (рис. 2.48):

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

Рис. 2.48. **Фільтр намірів**

Елемент `<action>` указує, що це *основна* точка входу в додаток. Елемент `<category>` указує, що цю операцію слід указати в засобі якості додатків системи (щоб користувачі могли запускати цю операцію).

Якщо додаток заплановано 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 у прикладі вище. В операціях, які не мають бути доступні для інших додатків, не слід включати фільтри намірів. Можна самостійно запустити такі операції за допомогою явних намірів.

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

Запуск операції. Для запуску іншої операції досить викликати метод `startActivity()`, передавши в нього об'єкт `Intent`, який описує операцію, що запускається. У намірі вказують або точну операцію для запуску, або описують тип операції, яку потрібно виконати (після чого система вибирає для вас підходящу операцію, яка може навіть перебувати в іншому додатку). Намір також може містити невеликий обсяг даних, які буде використовувати запущена операція.

Під час роботи з власним додатком найчастіше необхідно лише запустити прийнятну операцію. Для цього слід створити намір, що явно визначає необхідну операцію з допомогою назві класу. Далі наведено приклад запуску однією операцією іншої операції з назвою `SignInActivity` (рис. 2.49):

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

Рис. 2.49. **Одна операція запущена іншою операцією**

Однак у додатку також може знадобитися виконати деяку дію, наприклад, відправити лист, текстове повідомлення або оновити статус, використовуючи дані з операції. У цьому випадку в додатку можуть бути відсутні такі дії, тому можна скористатися операціями з інших додатків, наявних на пристрої, що можуть виконувати необхідні дії. Якраз у цьому разі наміри особливо корисні: можна створити намір, що описує необхідну дію, після чого система запускає його з іншої програми. За наявності декількох операцій, які можуть обробити намір, користувач може вибирати, яку з них слід використовувати. Наприклад, якщо користувачеві потрібно надати можливість відправити електронний лист, можна створити такий намір (рис. 2.50):

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

Рис. 2.50. **Створення наміру з відправлення листа**

Додатковий компонент `EXTRA_EMAIL`, доданий у намір, становить строковий масив адреси електронної пошти для відправлення листа. Коли поштова програма реагує на цей намір, вона зчитує додатково доданий строковий масив і поміщає наявні в ньому адреси в поле отримувача в вікні створення листа. Одночасно запускається операція поштової програми, а після того, як користувач завершить необхідні дії, відновлюється операція.

Запуск операції для отримання результату. У деяких випадках після запуску операції може знадобитися отримати результат. Для цього слід викликати метод `startActivityForResult()` (замість `startActivity()`). Щоб отримати результат після виконання наступної операції, потрібно реалізувати метод зворотного виклику `onActivityResult()`. Після завершення наступної операції вона повертає результат в об'єкті `Intent` у викликаний метод `onActivityResult()`.

Наприклад, користувачеві буде потрібно вибрати один із контактів, щоб операція могла виконати деякі дії з інформацією про цей контакт. Далі наведено приклад створення такого наміру й оброблення результату (рис. 2.51).

```
private void pickContact() {
    // Створити намір для "вибору" контакту, як визначено постачальником
контенту URI
    Intent intent = new Intent(Intent.ACTION_PICK, Contacts.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()) { // Вірно, якщо курсор не порожній
            int columnIndex =
                          cursor.getColumnIndex(Contacts.DISPLAY_NAME)
;
            String name = cursor.getString(columnIndex);
            // Зробити що-небудь з назвою зворотного контакту
        }
    }
}
```

Рис. 2.51. **Створення наміру з відбору одного з контактів**

У цьому прикладі демонструється базова логіка, якою слід керуватися під час використання методу `onActivityResult()` для оброблення результату виконання операції. Перша умова перевіряє чи можна пізнати запит, і якщо він успішний, то результат для `resultCode` буде `RESULT_OK`; також перевіряє 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`).

Одночасно `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 не слід явно завершувати операцію за допомогою цих методів. Система Android виконує таке управління сама, тому не потрібно завершувати власні операції. Виклик цих методів може негативно позначитися на очікуваній поведінці додатка. Їх слід використовувати виключно тоді, коли абсолютно не потрібно, щоб користувач повертався до цього екземпляра операції.

Управління життєвим циклом операцій. Управління життєвим циклом операцій шляхом реалізації методів зворотного виклику має важливе значення під час розроблення надійних і гнучких програм. На життєвий цикл операцій безпосередньо впливає його зв'язок з іншими операціями, завданнями та стеком переходів назад.

Існує всього три стани операції:

Відновлена. Операція виконується на передньому плані екрана та відображається для користувача. (Цей стан також іноді називається "Виконується").

Припинена. На передньому фоні виконується інша операція, яка відображається для користувача, однак перша операція як і раніше не прихована. Тобто над поточною операцією показується інша операція, частково прозора або що не займає повністю весь екран. Призупинена операція повністю активна (об'єкт `Activity` як і раніше знаходиться в пам'яті, у ньому зберігаються всі відомості про стан та інформація про елементи, і він також залишається пов'язаним із диспетчером вікон), проте в разі гострого браку пам'яті система може завершити її.

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, у ньому зберігаються всі відомості про стан та інформація про елементи, але об'єкт більше не пов'язаний із диспетчером вікон). Однак операція більше не видна користувачеві, і в разі нестачі пам'яті система може завершити її.

Якщо операцію припинено або повністю зупинено, система може очистити її з пам'яті шляхом завершення самої операції (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. Під час переходу операції з одного описаного стану до іншого повідомлення про це реалізуються через різні методи зворотного виклику. Усі методи зворотного виклику мають прив'язування, які можна визначити для виконання відповідної дії в разі зміни стану операції. Зазначена далі базова операція містить кожен з основних методів життєвого циклу (рис. 2.52).

```
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 призупинена)
    }
    @Override
    protected void onStop() {
      super.onStop();
      // Дія більше не відображається (вона тепер "зупинена")
    }
    @Override
    protected void onDestroy() {
      super.onDestroy();
      // Дія скоро буде видалена
    }
}
```

Рис. 2.52. **Методи життєвого циклу**

Примітка. Під час реалізації цих методів життєвого циклу завжди слід викликати реалізацію суперкласу, перш ніж виконувати будь-які дії, як показано у прикладах раніше.

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

Увесь життєвий цикл операції відбувається між викликами методів `onCreate()` та `onDestroy()`. Операція має виконати налаштування *глобального* стану (наприклад, визначення макета) в методі `onCreate()`, а потім звільнити всі ресурси, що залишилися в `onDestroy()`. Наприклад, якщо в операції є потік, що виконується у фоновому режимі, для завантаження даних по мережі, операція може створити такий потік у методі `onCreate()`, а потім зупинити його в методі `onDestroy()`.

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` у методі `onStart()` для відстеження змін, що впливають на інтерфейс користувача, 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.

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 піти у фоновий режим і виходити з нього, наприклад, метод `onPause()` викликається під час переходу пристрою у сплячий режим або появи діалогового вікна. Оскільки перехід до цього стану може виконуватися досить часто, код у цих двох методах має бути легким, щоб не допустити повільних переходів і не змушувати користувача чекати.

На рис. 2.53 ілюструються проходи та шляхи, які операція може пройти між станами. Прямокутниками позначено методи зворотного

виклику, що можна реалізувати для виконання дій між переходами операції з одного стану до іншого.

Ці ж методи життєвого циклу перелічено в табл. 2.5, у якій детально описано кожен метод зворотного виклику та вказано його місце в життєвому циклі операції загалом, включаючи відомості про те, чи може система завершити операцію після завершення методу зворотного виклику.

Рис. 2.53. **Життєвий цикл операції**

Таблиця 2.5

Зведені відомості про методи зворотного виклику життєвого циклу операції

Методи	Описи	Завершуючий	Наступний
1	2	3	4
`onCreate()`	Викликається під час першого створення операції. Тут необхідно налаштувати всі звичайні статичні елементи: створити уявлення, прив'язати дані та ін. Цей метод передає об'єкт Bundle, що містить попередній стан операції (якщо такий стан було зафіксовано раніше). За ним завжди слідує метод `onStart()`	Ні	`onStart()`
`onRestart()`	Викликається після зупинки операції безпосередньо перед її повторним запуском. За ним завжди слідує метод `onStart()`	Ні	`onStart()`
`onStart()`	Викликається безпосередньо перед тим, як операція стає видимою для користувача. За ним слідує метод `onResume()`, якщо операція переходить на передній план, або метод `onStop()`, якщо вона стає прихованою	Ні	`onResume()` або `onStop()`
`onResume()`	Викликається безпосередньо перед тим, як операція починає взаємодію з користувачем. На цьому етапі операція перебуває нагорі стека операцій, і в неї надходять дані, що вводяться користувачем. За ним завжди слідує метод `onPause()`	Ні	`onPause()`
`onPause()`	Викликається, коли система збирається відновити іншу операцію. Цей метод зазвичай використовується для запису незбережених змін у постійне місце зберігання даних, зупинки анімацій та інших елементів, які можуть використовувати ресурси ЦП та ін. Тут украй важлива оперативність, оскільки наступну операцію не буде відновлено до тих пір, поки її не буде повернено на передній план. За ним слідує або метод `onResume()`, якщо операція повертається на передній план, або метод `onStop()`, якщо операція стає прихованою для користувача	Так	`onResume()` або `onStop()`

Закінчення табл. 2.5

1	2	3	4
`onStop()`	Викликається в разі, коли операція більше не відображається для користувача. Це може статися через те, що операцію знищено, або, зважаючи на відновлення над неї іншої операції (наявної або нової). За ним слідує або метод `onRestart()`, якщо операція відновлює взаємодія з користувачем, або метод `onDestroy()`, якщо операція переходить у фоновий режим	Так	`onRestart()` або `onDestroy()`
`onDestroy()`	Викликається перед тим, як операцію буде знищено. Це фінальний виклик, який отримує операція. Його можна викликати або через завершення операції (виклик методу `finish()`, або через тимчасове знищення системою цього примірника операції, із метою звільнити місце. Щоб розрізнити ці два сценарії, використовується метод `isFinishing()`	Так	Нічого

У стовпці "Завершуючий" указують, 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 у цьому стовпці вказано "Так": (`onPause()`, `onStop()` та `onDestroy()`). Оскільки метод `onPause()` є першим із 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) у сховище постійних даних. Однак слід уважно підходити до вибору інформації, яку необхідно зберегти під час виконання методу `onPause()`, оскільки будь-яке блокування процедур у цьому методі може викликати блокування переходу до наступної операції та гальмувати роботу користувача.

Методи, для яких у стовпці "Завершуючий" вказано "Ні", захищають процес, що містить операцію, від завершення відразу з моменту їхнього виклику. Тому завершити операцію можна в період між поверненням

`onPause()` і викликом `onResume()`. Його не вдасться завершити, поки знову не буде викликано і повернено `onPause()`.

Примітка. Операцію, яку технічно неможливо завершити відповідно, до визначення в табл. 2.5, 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` до того, як і раніше, перебуває в пам'яті та вся інформація про її елементи й поточний стан, як і раніше, є активною. Тому будь-які зміни, які вносяться користувачем в операції, зберігаються, і коли операція повертається на передній план (коли вона *відновлюється*), ці зміни залишаються в цьому об'єкті.

Однак, коли система знищує операцію із метою відновлення пам'яті, об'єкт `Activity` знищується, у результаті чого системі не вдається просто відновити 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 та створила її повторно, 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 операції: `onSaveInstanceState()`.

Перш ніж зробити операцію доступною для знищення, система викликає метод `onSaveInstanceState()`. Система передає в цей метод об'єкт `Bundle`, у якому можна зберегти інформацію про стан операції у вигляді пар "назва-значення", використовуючи для цього такі методи, як `putString()` та `putInt()`. Потім, якщо система завершує процес вашого додатка й користувач повертається до операції, система повторно створює операцію та 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 та 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. 2.54).

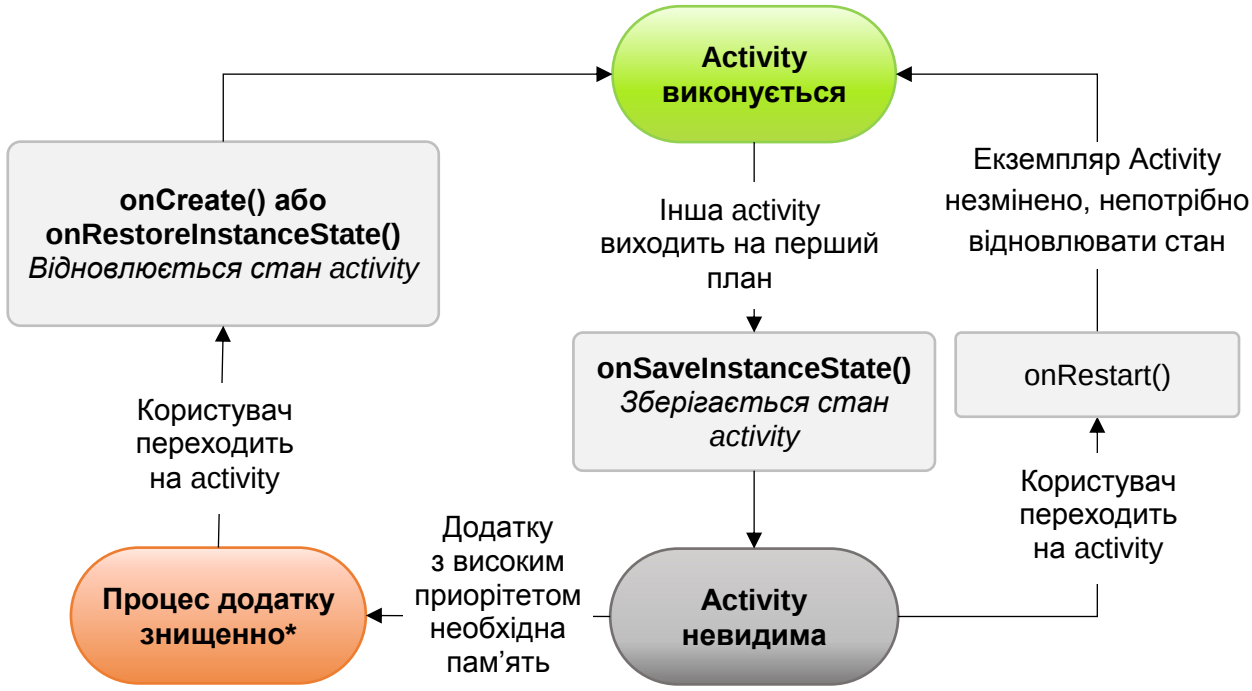

Рис. 2.54. **Два способи повернення операції**

На рис. 2.54. можна побачити два способи повернення операції до відображення для користувача в незміненому стані: знищення операції з подальшим її повторним створенням, коли операція має відновити свій раніше збережений стан, або зупинка операції та її подальше відновлення в незміненому стані.

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

Однак, навіть якщо нічого не робити і не реалізувати метод `onSaveInstanceState()`, частина статків операції відновлюється реалізацією за замовчуванням методу `onSaveInstanceState()` класу `Activity`. Зокрема, реалізація за замовчуванням викликає відповідний метод `onSaveInstanceState()` для кожного об'єкта `View` у макеті, завдяки чому кожна вистава 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 та відновлюються за повторного створення операції. Наприклад, віджет `EditText` зберігає будь-який текст, який самі ввели, а віджет `CheckBox` зберігає інформацію про те, чи було встановлено прапорець. Потрібно 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 слід задати для атрибута `android:saveEnabled` значення `false` або викликати метод `setSaveEnabled()`. Зазвичай відключати збереження такої інформації є необхідним, проте це може знадобитися у випадках, коли відновити стан інтерфейсу користувача операції необхідно іншим чином.

Незважаючи на те що реалізація методу `onSaveInstanceState()` за замовчуванням дозволяє зберегти корисну інформацію про інтерфейс користувача операції, як і раніше може знадобитися визначити 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 користувача, проте елементи, що містять ці значення інтерфейсу користувача, за замовчуванням не було відновлено).

Оскільки реалізація методу `onSaveInstanceState()` за замовчуванням дозволяє зберегти стан інтерфейсу користувача, в разі, якщо перевизначити метод із метою зберегти додаткову інформацію про стан, перед виконанням будь-яких дій завжди можна викликати реалізацію суперкласу для методу `onSaveInstanceState()`. Точно так же реалізацію суперкласу `onRestoreInstanceState()` слід викликати в разі її перевизначення, щоб реалізація за замовчуванням могла зберегти стан уявлень.

Примітка. Оскільки виклик методу `onSaveInstanceState()` не гарантовано, слід використовувати його тільки для запису перехідного стану операції – ніколи не застосовувати його для зберігання постійних даних. Замість цього потрібно використовувати метод `onPause()` для збереження постійних даних (наприклад, тих, які слід зберегти в базу даних), коли користувач залишає операцію.

Відмінний спосіб перевірити можливість додавання відновлювати свій стан – це просто повернути пристрій для зміни орієнтації екрана. У разі зміни орієнтації екрана система знищує та повторно створює операцію, щоб застосувати альтернативні ресурси, які можуть бути доступні для нової конфігурації екрана. Тільки з однієї цієї причини вкрай важливо, щоб операція могла повністю відновлювати свій стан за її повторного створення, оскільки користувачі постійно працюють із додатками в різних орієнтаціях екрана.

Оброблення змін у конфігурації. Деякі конфігурації пристроїв можуть змінюватися в режимі виконання (наприклад, орієнтація екрана, доступність клавіатури та 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 надали (наприклад, різні макети для різних орієнтацій та екранів різних розмірів).

Якщо операцію розроблено належним чином і належним чином вона підтримує перезапуск після зміни орієнтації екрана та відновлення свого стану, як описано раніше, додаток можна вважати більш стійким до інших непередбачених подій у життєвому циклі операції.

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

Погодження операцій. Коли одна операція запускає іншу, у життєвих циклах обох із них відбувається перехід з одного стану до іншого. Перша операція припиняється та завершується (проте її не буде зупинено, 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 до тих пір, поки не буде створено другу операцію. Навпаки, процес запуску другої операції накладається на процес зупинки першої операції.

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

1. Виконується метод `onPause()` операції А.

2. Послідовно виконують методи `onCreate()`, `onStart()` та `onResume()` операції Б (тепер для користувача відображається операція Б.).

3. 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 до іншої. Наприклад, якщо після зупинки першої операції потрібно виконати запис в базу даних, щоб наступна операція могла вважати їх, то цей запис слід виконати під час виконання методу `onPause()`, а не під час виконання методу `onStop()`.

2.6.2. Фрагменти

Фрагмент (клас `Fragment`) становить поведінку або частину інтерфейсу користувача в операції (клас `Activity`). Розробник може об'єднати кілька фрагментів в одну операцію для побудови багатопанельного фрагмента інтерфейсу користувача та 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 укладеної операції, яку можна багаторазово використовувати в різних операціях.

Фрагмент завжди має бути вбудовано в операцію, і на його життєвий цикл безпосередньо впливає життєвий цикл операції. Наприклад, коли операцію припинено, у тому ж стані перебувають і всі фрагменти всередині неї, а коли операція знищується, знищуються і всі фрагменти. Однак поки операція виконується (це відповідає стану *відновлена* життєвого циклу), можна маніпулювати кожним фрагментом незалежно, наприклад, додавати або видаляти їх. Коли розробник виконує такі транзакції із фрагментами, він може також додати їх у стек переходів, яким управляє операція. Кожен елемент стека переходів назад в операції є записом виконаної транзакції з фрагментом. Стек переходів назад дозволяє користувачеві звернути транзакцію із фрагментом (виконати навігацію у зворотному напряму), натискаючи кнопку *Назад*.

Коли фрагмент додано як частину макета операції, він перебуває в об'єкті `ViewGroup` усередині ієрархії уявлень операції та визначає власний макет уявлень. Розробник може вставити фрагмент у макет операції двома способами. Для цього слід оголосити фрагмент у файлі макета операції як елемент `<fragment>` або додати його в наявний об'єкт `ViewGroup`

у коді програми. Утім, фрагмент не зобов'язаний бути частиною макета операції. Можна використовувати фрагмент без інтерфейсу як невидимий робочий поток операції.

У цьому документі показано, як побудувати додаток, що використовує фрагменти. Зокрема, обговорено, як фрагменти можуть підтримувати свій стан, коли їх додають у стек переходів назад операції, використовувати події спільно з операцією та іншими фрагментами всередині неї, виводити дані в рядок дій операції тощо.

Філософія проектування

Фрагменти вперше з'явилися в Android-версії 3.0 (API рівня 11), головним чином, для забезпечення більшої динамічності та гнучкості інтерфейсу користувача на великих екранах, наприклад, у планшетів. Оскільки екрани планшетів набагато більші, ніж у смартфонів, вони надають більше можливостей для об'єднання та переставлення компонентів для інтерфейсу користувача. Фрагменти дозволяють робити це, позбавляючи розробника від необхідності управляти складними змінами в ієрархії уявлень. Розбиваючи макет операції на фрагменти, розробник має можливість модифікувати зовнішній вигляд операції в ході виконання та зберігати ці зміни у стеці переходів, яким управляє 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 й управляє власними подіями призначеними для користувача. Таким чином, замість застосування однієї операції для вибору статті, а іншої – для читання статей, користувач може вибрати статтю та читати її в межах однієї операції, як на планшеті, зображеному на рис. 2.55.

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 має передбачити повторне використання фрагмента та не допускати, щоб один фрагмент безпосередньо маніпулював іншим. Це особливо важливо, тому що модульність фрагментів дозволяє змінювати їхнє 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 розподілі фрагментів для надання однопанельного інтерфейсу користувача, якщо розробнику не вдається помістити більше одного фрагмента в одну операцію.

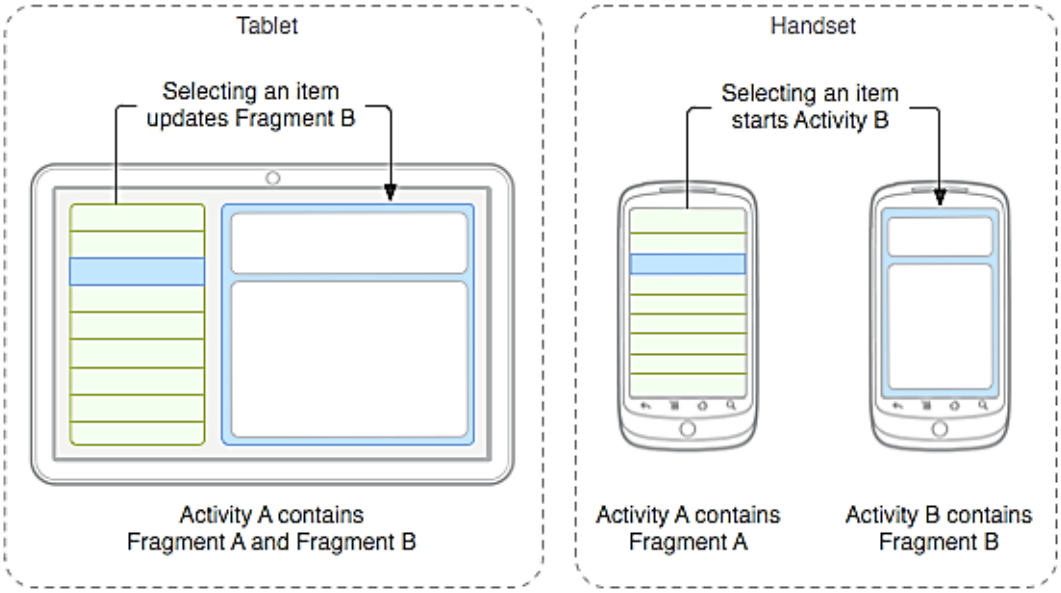

Рис. 2.55. **Об'єднані фрагменти всередині однієї операції**

На рис. 2.55 можна побачити приклад того, як два модулі інтерфейсу користувача, визначені фрагментами, можуть бути об'єднані всередині однієї операції для роботи на планшетах, але розподілені на смартфонах.

Слід повернутися до прикладу з новинним додатком. Він може мати два фрагменти, вбудованих в *Операцію А*, коли виконується на пристрої планшетного формату. Водночас на екрані смартфона недостатньо місця для обох фрагментів, і тому *Операція А* містить тільки фрагмент зі списком статей. Коли користувач вибирає статтю, запускається *Операція В*, що містить другий фрагмент для читання статті. Таким чином, додаток підтримує як планшети, так і смартфони, завдяки повторному використанню фрагментів у різних поєднаннях, як показано на рис. 2.56.

Створення фрагмента

Для створення фрагмента необхідно створити підклас класу `Fragment` (або його наявного підкласу). Клас `Fragment` має код, багато в чому схожий на код `Activity`. Він містить такі методи зворотного виклику, аналогічні методам операції, як `onCreate()`, `onStart()`, `onPause()` та `onStop()`. Життєвий цикл фрагмента наведено на рис. 2.56.

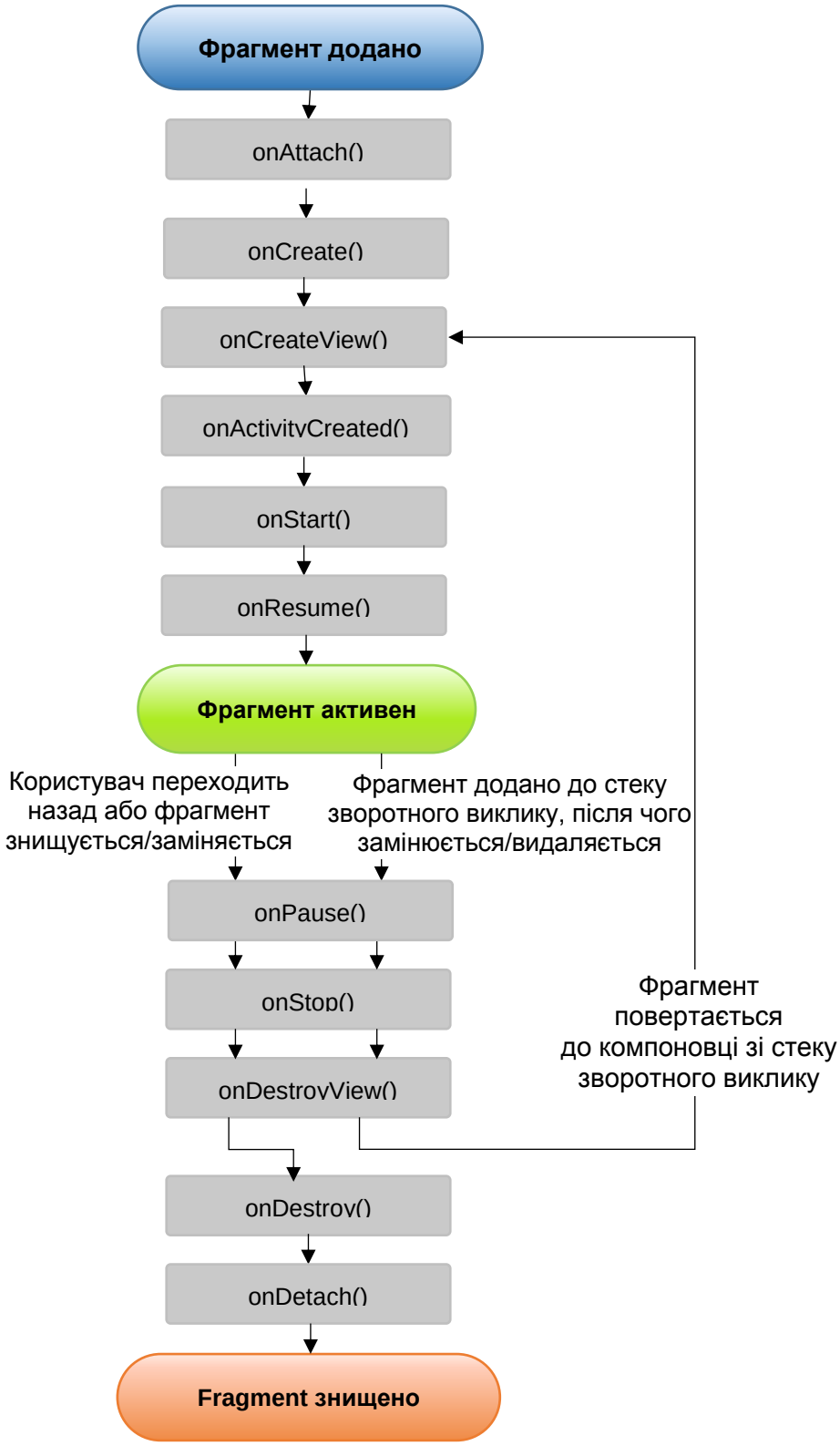

Рис. 2.56. **Життєвий цикл фрагмента** (під час виконання операції)

На практиці, якщо потрібно перевести наявну програму Android так, щоб у ній використовували фрагменти, досить просто перемістити код із методів зворотного виклику операції у відповідні методи зворотного виклику фрагмента.

Переважно, необхідно реалізувати такі методи життєвого циклу:

`onCreate()`: система викликає цей метод, коли створює фрагмент. У своїй реалізації розробник має ініціалізувати ключові компоненти фрагмента, які необхідно зберігати, коли фрагмент перебуває у стані паузи або відновлений після зупинки;

`onCreateView()`: система викликає цей метод під час першого відображення фрагмента інтерфейсу користувача на дисплеї. Для промальовування фрагмента інтерфейсу користувача слід повернути з цього методу об'єкт `View`, який є кореневим у макеті фрагмента. Якщо фрагмент не має інтерфейсу користувача, можна повернути null;

`onPause()`: система викликає цей метод як першу вказівку на те, що користувач залишає фрагмент (це не завжди означає знищення фрагмента). Зазвичай саме в цей момент необхідно фіксувати всі зміни, які мають бути збережені за межами поточного сеансу роботи користувача (оскільки користувач може не повернутися назад);

У більшості додатків для кожного фрагмента має бути реалізовано, 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`: відображення переміщуваного діалогового вікна. Використання цього класу для створення діалогового вікна є хорошою альтернативою допоміжних методів діалогового вікна у класі `Activity`. Справа в тому, що він дає можливість уставити діалогове вікно фрагмента в керований операцією стек переходів назад для фрагментів, що дозволяє користувачеві повернутися до закритого фрагмента;

`ListFragment`: відображення списку елементів, керованих адаптером (наприклад, `SimpleCursorAdapter`), аналогічно класу `ListActivity`. Цей клас надає кілька методів для управління списком уявлень, наприклад, метод зворотного виклику `onListItemClick()` для оброблення натискань;

`PreferenceFragment`: відображення ієрархії 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 інтерфейсу користувача операції, одночасно він додає в операцію свій макет.

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

Примітка. Якщо фрагмент є підкласом класу `ListFragment`, реалізація за замовчуванням повертає клас `ListView` із методу `onCreateView()`, так що реалізовувати його немає необхідності.

Щоб повернути макет із методу `onCreateView()`, можна виконати його роздування з *ресурсу макета*, визначеного у XML-файлі. Із цією метою метод `onCreateView()` надає об'єкт `LayoutInflater`.

Наприклад, код підкласу класу `Fragment`, завантажує макет із файла `example_fragment.xml`, що можна мати такий вигляд (рис. 2.57):

```
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);
  }
}
```

Рис. 2.57. **Код підкласу класу `Fragment`, що завантажує макет із файла `example_fragment.xml`**

Параметр `container`, переданий методу `onCreateView()`, є батьківським класом `ViewGroup` (із макета операції), у який буде вставлено 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, роздування якого слід виконати;
- об'єкт класу `ViewGroup`, який має стати батьківським для макета після роздування. Передача параметра `container` необхідна для того, щоб система змогла застосувати параметри макета до кореневого подання роздутого макета, який визначається батьківським уявленням, до якого направляється макет;

- логічне значення, яке показує, чи слід прикріпити макет до об'єкта `ViewGroup` (другий параметр) під час роздування (у цьому випадку це false, тому що система вже вставляє роздутий макет в об'єкт `container`, і передача значення true створила б зайву групу уявлення в остаточному макеті).

Побачивши, як створювати фрагмент, що надає макет, тепер необхідно додати фрагмент в операцію.

Додавання фрагмента в операцію. Переважно, фрагмент додає частину інтерфейсу користувача в операцію, і цей інтерфейс вбудовується в загальну ієрархію уявлень операції. Розробник може додати фрагмент у макет операції двома способами:

оголосивши фрагмент у файлі макета операції.

У цьому випадку можна вказати властивості макета для фрагмента, як ніби він є уявленням. Наприклад, файл макета операції із двома фрагментами може мати такий вигляд (рис. 2.58):

```
<?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>
```

Рис. 2.58. **Файл макета операції із двома фрагментами**

Атрибут `android:name` в елементі `<fragment>` визначає клас `Fragment`, екземпляр якого створено 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 має унікальний ідентифікатор, який система зможе використовувати для відновлення фрагмента в разі перезапуску операції. (Що стосується розробника, він може використовувати цей ідентифікатор для захоплення фрагмента, із метою виконання транзакцій із ним, наприклад, щоб видалити його). Надати ідентифікатор фрагмента можна трьома способами:

указавши атрибут `android:id` з унікальним ідентифікатором;

указавши атрибут `android:tag` з унікальною рядком;

нічого не робити, щоб система використовувала ідентифікатор контейнерного уявлення;

програмним чином, додавши фрагмент в наявний об'єкт `ViewGroup`.

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

Для виконання транзакцій із фрагментами всередині операції (таких як додавання, видалення або заміна фрагмента) необхідно використовувати API-інтерфейси із `FragmentTransaction`. Екземпляр класу `Fragment Transaction` можна отримати від об'єкта `Activity` таким чином (рис. 2.59):

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

Рис. 2.59. **Отримання екземпляра класу FragmentTransaction від об'єкта Activity**

Після цього можна додати фрагмент методом `add()`, указавши доданий фрагмент та уявлення, у яке він має бути доданий. Наприклад (рис. 2.60):

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

Рис. 2.60. **Додавання фрагмента**

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

Виконавши зміни за допомогою `FragmentTransaction`, необхідно викликати метод `commit()`, щоб вони були застосовані.

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

Щоб додати фрагмент без інтерфейсу користувача, слід додати фрагмент з операції, використовуючи метод `add(Fragment, String)` (передавши йому унікальний *строковий тег* для фрагмента замість ідентифікатора уявлення). Фрагмент буде додано, але, оскільки він не пов'язаний із поданням до макета операції, він не буде приймати виклик методу `onCreateView()`. Тому в реалізації цього методу немає необхідності.

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

Приклад операції, що використовує фрагмент як фоновий потік, без інтерфейсу користувача, наведено у зразку коду `FragmentRetainInstance.java`, входить до зразків у 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()` із коду операції.

Далі вказано дії, які дозволяє виконати `FragmentManager`:

отримувати фрагменти, наявні в операції, за допомогою метода `findFragmentById()` (для фрагментів, які надають інтерфейс користувача в макеті операції) чи `findFragmentByTag()` (як для фрагментів, які 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 у стеку переходів назад за допомогою методу `addOnBackStackChangedListener()`.

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

Як було показано раніше, можна використовувати клас `FragmentManager` для відкриття `FragmentTransaction`, що дозволяє виконувати транзакції із фрагментами, наприклад, додавання та 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 вносять до операції, називають транзакцією. Її можна виконати за допомогою API-інтерфейсів у `FragmentTransaction`. Кожну транзакцію можна зберегти у стоку переходів тому, яким управляє операція. Це дозволить користувачеві переміщатися назад щодо змін у фрагментах (аналогічно переміщенню назад за операціями).

Екземпляр класу `FragmentTransaction` можна отримати від `FragmentManager`, наприклад, так (рис. 2.61):

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

Рис. 2.61. **Отримання екземпляра класу `FragmentTransaction` від FragmentManager**

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

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

Наприклад, такий код демонструє, як можна замінити один фрагмент іншим, зберігши одночасно попередній стан у стеку переходів назад (рис. 2.62):

```
// Створити новий фрагмент і транзакцію
Fragment newFragment = new ExampleFragment();
FragmentTransaction transaction =
                                getFragmentManager().beginTransaction();

// Замінити всі, що є в поданні 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();
```

Рис. 2.62. **Заміна одного фрагмента іншим**

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

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

Порядок додавання змін до об'єкта `FragmentTransaction` не грає ролі за такими винятками:

- метод `commit()` має бути викликано в останню чергу;
- якщо в один контейнер додано кілька фрагментів, то порядок їхнього додавання визначає порядок, у якому вони з'являються в ієрархії видів.

Під час виконання транзакції, що видаляє фрагмент, якщо не викликати метод `addToBackStack()`, під час фіксації транзакції фрагмент знищується, і користувач утрачає можливість повернутися до нього.

Водночас, якщо викликати `addToBackStack()` під час видалення фрагмента, фрагмент перейде у стан *зупинки* і буде відновлений, коли користувач повернеться до нього.

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

Виклик методу `commit()` не призводить до негайного виконання транзакції. Метод запланує її виконання в потоці інтерфейсу користувача операції (у *головному* потоці), як тільки в потоку з'явиться можливість для цього. Утім, за потреби можна викликати `executePending Transactions()` із потоку інтерфейсу користувача, щоб транзакції, заплановані методом `commit()`, було виконано негайно. Переважно, у цьому немає необхідності, 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 втрачено, якщо знадобиться відновити операцію. У ситуаціях, у яких утрата фіксації не критична, слід викликати `commitAllowingStateLoss()`.

Взаємодія з операцією

Хоча `Fragment` реалізовано як об'єкт, незалежний від класу `Activity`, і може бути використано 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 завдання, як пошук уявлення в макеті операції (рис. 2.63):

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

Рис. 2.63. **Звернення до примірника Activity за допомогою методу `getActivity()`**

Аналогічним чином операція може викликати методи фрагмента, отримавши посилання на об'єкт `Fragment` від `FragmentManager` за допомогою методу `findFragmentById()` чи `findFragmentByTag()`. Наприклад (рис. 2.64):

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

Рис. 2.64. **Отримання посилання на об'єкт `Fragment` від `FragmentManager`**

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 та 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.

Нехай, наприклад, у новинного додатка є два фрагменти в одній операції: один для відображення списку статей (фрагмент A), а інший – для відображення статті (фрагмент B). Тоді фрагмент A має повідомляти операції про те, що обрано пункт списку, щоб вона могла повідомити фрагменту B про необхідність відобразити статтю. У цьому випадку інтерфейс `OnArticleSelectedListener` оголошено у фрагменті A (рис. 2.65):

```
public static class FragmentA extends ListFragment {
    ...
    // Container Activity повинен реалізовувати цей інтерфейс
    public interface OnArticleSelectedListener {
        public void onArticleSelected(Uri articleUri);
    }
    ...
}
```

Рис. 2.65. **Оголошення інтерфейсу `OnArticleSelectedListener`**

Тоді операція, яка містить цей фрагмент, реалізує інтерфейс `OnArticleSelectedListener` та перевизначить метод `onArticleSelected()`, щоб сповіщати фрагмент B про подію, що виходить від фрагмента A. Щоб контейнерна операція напевно реалізувала цей інтерфейс, метод зворотного виклику `onAttach()` у фрагменті A (який система викликає під час додавання фрагмента в операцію) створює екземпляр класу `OnArticleSelectedListener`, виконавши приведення типу об'єкта `Activity`, який передано методу `onAttach()` (рис. 2.66):

```
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");
        }
    }
    ...
}
```

Рис. 2.66. **Перевизначення методу `onAttach()`**

Якщо операція не реалізувала інтерфейс, фрагмент генерує виняток `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 щораз, коли користувач натискає елемент списку, система викликає `onListItemClick()` у фрагменті. Цей метод, у свою чергу, викликає метод `onArticleSelected()`, щоб використовувати подію спільно з операцією (рис. 2.67):

```
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 provider Uri
    Uri noteUri = ContentUris.withAppendedId(ArticleColumns.CONTENT_URI, id);
```

Рис. 2.67. **Визначення `onListItemClick()` у фрагменті**

```
    // Send the event and Uri to the host activity
        mListener.onArticleSelected(noteUri);
    }
    ...
}
```

Закінчення рис. 2.67

Параметр `id`, передаваний 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 у рядок дій. Фрагменти можуть додавати пункти меню в Меню варіантів операції (і, отже, у *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 наявних. Крім того, фрагмент приймає зворотні виклики методу `onOptionsItemSelected()`, коли користувач вибирає пункт меню.

Розробник може також зареєструвати уявлення 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 першої приймає відповідний зворотний виклик, коли користувач вибирає пункт меню. Якщо наявна в операції реалізація зворотного виклику по події "обраний пункт меню" не виконує жодних обраних пунктів, подія передається методу зворотного виклику у фрагменті. Це справедливо для Меню варіантів і контекстних меню.

Докладні відомості щодо меню дивіться 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 із трьох станів:

відновлений. Фрагмент видно під час виконання операції;

призупинений. На передньому плані виконується та перебуває у фокусі інша операція, але операція, яка містить цей фрагмент, як і раніше видна (операція переднього плану частково прозора або не займає весь екран);

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

Найвизначніша відмінність у ході життєвого циклу між операцією та фрагментом полягає у принципах їхнього 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*). Водночас, фрагмент поміщається у стек переходів назад, керований операцією, тільки коли розробник явно запросить збереження конкретного екземпляра, викликавши метод `addToBackStack()` під час транзакції, що видаляє фрагмент.

В іншому разі управління життєвим циклом фрагмента дуже схоже на управління життєвим циклом операції. Тому практичні рекомендації з *управління життєвим циклом операцій* застосовні й до фрагментів. До того ж розробнику необхідно розуміти, як життєвий цикл операції впливає на життєвий цикл фрагмента.

Увага! Якщо виникне необхідність в об'єкті `Context` усередині об'єкта класу `Fragment`, можна викликати метод `getActivity()`. Однак розробник

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

Погодження з життєвим циклом операції. Життєвий цикл операції, що містить фрагмент, безпосереднім чином впливає на життєвий цикл фрагмента, так що 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 та знищення фрагмента інтерфейсу користувача. Ось ці методи:

`onAttach()`. Викликається, коли фрагмент зв'язується з операцією (йому передається об'єкт `Activity`);

`onCreateView()`. Викликається для створення ієрархії уявлень, пов'язаної з фрагментом;

`onActivityCreated()`. Викликається, коли метод `onCreate()`, належить операції, повертає управління;

`onDestroyView()`. Викликається під час видалення ієрархії уявлень, пов'язаної з фрагментом;

`onDetach()`. Викликається під час розриву зв'язку фрагмента з операцією.

Залежність життєвого циклу фрагмента від операції, що містить його, проілюстровано на рис. 2.68. 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 усередині цієї операції приймає всього лише метод зворотного виклику `onActivityCreated()`.

Коли операція переходить до стану "відновлена", можна вільно додавати в неї фрагменти та видаляти їх. Таким чином, життєвий цикл фрагмента може бути незалежно змінено, тільки поки операція залишається у стані "відновлена".

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

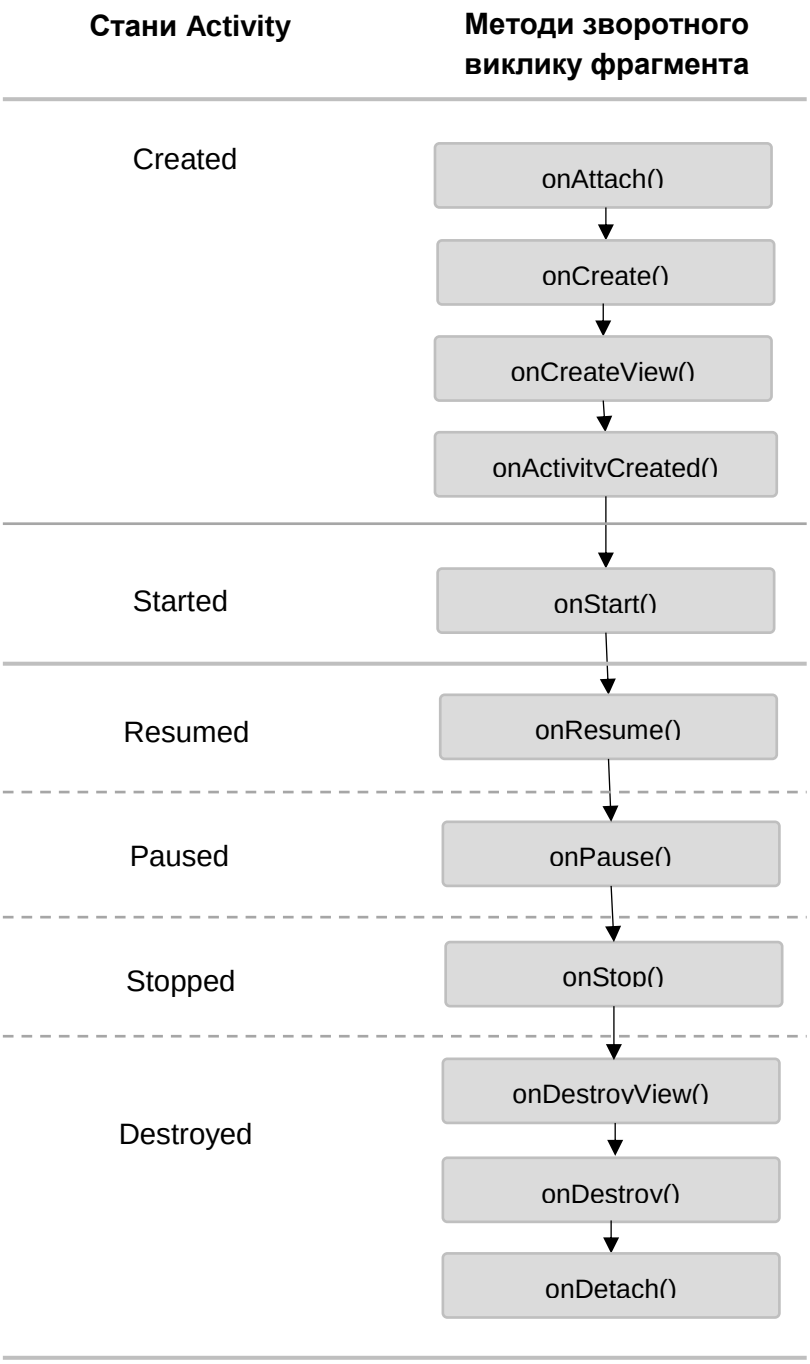

Рис. 2.68. **Вплив життєвого циклу операції на життєвий цикл фрагмента**

Приклад

Щоб підсумувати все сказане в цьому документі, слід розглянути приклад операції, що використовує два фрагменти для створення макета із двома панелями. Операція, код якої наведено далі, містить 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 у файлі `FragmentLayout.java`.

Головна операція застосовує макет звичайним способом, у методі `onCreate()` (рис. 2.69):

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

Рис. 2.69. **Метод `onCreate()`**

Тут застосовують макет `fragment_layout.xml` (рис. 2.70):

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

Рис. 2.70. **Макет `fragment_layout.xml`**

Користуючись цим макетом, система створює екземпляр класу `TitlesFragment` (список п'єс), як тільки операція завантажить макет. Одночасно об'єкт `FrameLayout` (у якому буде перебувати фрагмент із коротким змістом) займає місце у правій частині екрана, але спочатку залишається порожнім. Як буде показано далі, фрагмент не поміщається у `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 раніше макет використовується тільки за альбомної орієнтації екрана та зберігається у файлі `res/layout-land/fragment_layout.xml`.

Коли ж пристрій перебуває у книжковій орієнтації, система застосовує макет, наведений на рис. 2.71, який зберігається у файлі `res/layout/fragment_layout.xml`:

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

Рис. 2.71. **Макет** `fragment_layout`

У цьому макеті наявний тільки об'єкт `TitlesFragment`. Це означає, що за книжкової 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 у класах фрагмента. Спочатку йде код класу `TitlesFragment` (рис. 2.72; 2.73), який відображає список п'єс Шекспіра. Цей фрагмент є розширенням класу `ListFragment` і використовує його функції для виконання основної роботи зі списком.

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

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

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

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

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

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

        if (mDualPane) {
        // У двоканальному режимі панелі, уявлення списку виділяє
        //вибраний елемент.
            getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
        // Переконаймося, що наш інтерфейс користувача знаходиться у
        // правильному стані.
            showDetails(mCurCheckPosition);
        }
    }

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

Рис. 2.72. **Код класу `TitlesFragment`. Частина 1**

```
@Override
public void onListItemClick(ListView l, View v, int position, long id) {
  showDetails(position);
}
/**
* Допоміжна функція для відображення деталей обраного елемента або
* відображення фрагмента на місці в поточному інтерфейсі або
* запуск цілої нової дії, у який вона відображається.
*/
void showDetails(int index) {
  mCurCheckPosition = index;

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

    // Перевіряємо, який фрагмент відображається в цей момент, за
    // потреби 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 наявний фрагмент
    // із 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);
  }
}}
```

Рис. 2.73. **Код класу `TitlesFragment`. Частина 2**

Другий фрагмент, `DetailsFragment`, відображає короткий зміст п'єси, обраної у списку `TitlesFragment` (рис. 2.74):

```
public static class DetailsFragment extends Fragment {
/**
* Створіть новий екземпляр DetailsFragment, ініціалізується,
* щоб показати текст у "index". */
    public static DetailsFragment newInstance(int index) {
        DetailsFragment f = new DetailsFragment();

        // Індекс постачання в якості аргументу
        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) {
          // У нас є різні макети, і в одному з них кадр,
          // що містить цей фрагмент, не існує. Фрагмент
          // може бути створений з його збереженого стану,
          // але немає підстав створити його ієрархію перегляду,
          // тому що вона не відображатиметься. Зауважте,
          // що це не потрібно - ми могли б просто запустити код нижче,
          // де ми створимо і повернемо ієрархію перегляду;
          // це ніколи не буде використовуватися
          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;
    }
}
```

Рис. 2.74. **Код класу** `DetailsFragment`

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

Далі йде код класу `DetailsActivity` (рис. 2.75), який усього лише містить об'єкт `DetailsFragment` для відображення короткого змісту обраної п'єси на екрані у книжковій орієнтації:

```
public static class DetailsActivity extends Activity {

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

    if (getResources().getConfiguration().orientation
                == Configuration.ORIENTATION_LANDSCAPE) {
      // Якщо екран зараз знаходиться в альбомному режимі, ми можемо
показати його
      діалогове вікно відповідно зі списком, тому ми не потребуємо цієї
дії.
      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();
        }
    }
}
```

Рис. 2.75. **Код класу** `DetailsActivity`

Зверніть увагу, що в альбомної конфігурації ця операція самостійно завершується, щоб головна операція могла прийняти управління та відобразити фрагмент `DetailsFragment` поряд із фрагментом `TitlesFragment`.

Це може статися, якщо користувач запустить операцію `DetailsActivity` у книжковій орієнтації екрана, а потім переверне пристрій на режим пейзажу (у результаті чого поточну операцію буде перезапущено).

2.7. 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 в декількох аспектах, в основному їх використовують у 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 усі інші необхідні дані.

Якщо після завершення операції від неї потрібно отримати результат, слід викликати метод `startActivityForResult()`. Операція отримає результат у вигляді окремого об'єкта `Intent` у 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'язування 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()`.

2.7.1. Типи об'єктів Intent

Є два типи об'єктів Intent:

Явні об'єкти Intent вказують компонент, який потрібно запустити, за назвою (повна назва класу). Явні об'єкти Intent зазвичай використовують для запуску компонента із власного додатка, оскільки відомо назву класа 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 назву конкретного компонента. Замість цього вони загалом 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, у якому таку функцію передбачено.

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

1. Операція A створює об'єкт `Intent` з описом дії та передає його до методу `startActivity()`.

2. Система Android шукає у всіх додатках фільтри Intent, які відповідають цьому об'єкту `Intent`.

3. Коли додаток із відповідним фільтром знайдено, система запускає відповідну операцію (Операція B), викликавши її метод `onCreate()` і передавши йому об'єкт `Intent` (рис. 2.76).

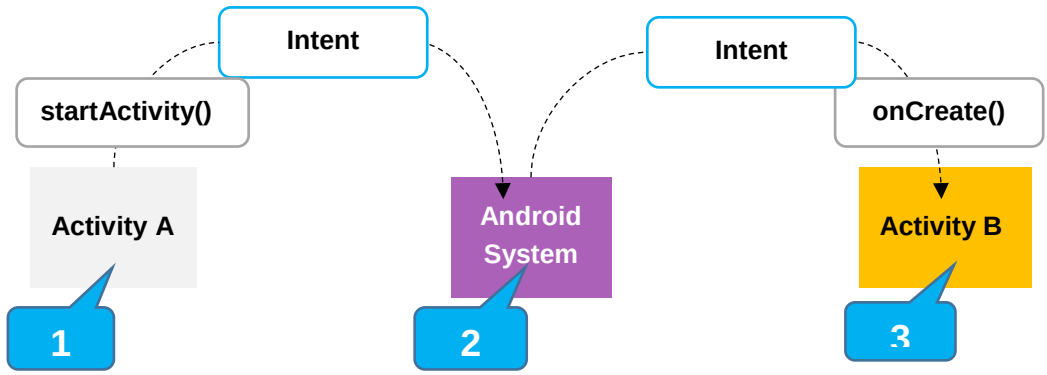

Рис. 2.76. **Процес передачі неявного об'єкта `Intent` по системі для запуску іншої операції**

Коли створено неявний об'єкт `Intent`, система Android знаходить відповідний компонент шляхом порівняння вмісту об'єкта `Intent` із 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.

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

Увага! Із метою забезпечення безпеки програми завжди використовувати явний об'єкт `Intent` під час запуску `Service` і не оголошувати фільтри Intent для своїх служб. Запуск служб за допомогою неявних об'єктів Intent є ризикованим із точки зору безпеки, оскільки не можна бути абсолютно впевненим, яка служба відреагує на такий об'єкт `Intent`, а користувач не може бачити, яка служба запускається. Починаючи з Android 5.0 (рівень API 21) система викликає виключення під час виклику методу `bindService()` за допомогою неявного об'єкта `Intent`.

2.7.2. Створення об'єкта Intent

Об'єкт `Intent` містить інформацію, на підставі якої система Android визначає, який компонент потрібно запустити (наприклад, точна назва компонента або категорія компонентів, які мають отримати цей об'єкт Intent), а також відомості, необхідні компоненту-отримувачу, щоб належним чином виконати дію (а саме, виконувану дію та дані, із якими його потрібно виконати).

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

Назва компонента

Назва 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 назвою. За відсутності назви компонента об'єкт `Intent` є **неявним**, а система визначає, який компонент отримає цей об'єкт `Intent` за іншими відомостями, які в ньому містяться (наприклад, за дією, даними та категоріями – див. Опис далі). Тому, якщо потрібно запустити певний компонент зі свого програмного додатка, слід указати його назву.

Примітка. Під час запуску `Service` слід **завжди вказувати назву компонента**. В іншому випадку не можна бути абсолютно впевненим у тому, яка служба відреагує на об'єкт Intent, а користувач не може бачити, яка служба запускається.

Це поле об'єкта Intent є об'єктом `ComponentName`, який можна вказати за допомогою повної назви класу цільового компонента, включаючи назву пакета програми. Наприклад, `com.example.ExampleActivity`. Задати назву компонента можна за допомогою методів `setComponent()`, `setClass()`, `setClassName()` або конструктора `Intent`.

Дія

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

Під час видачі об'єктів `Intent` із широкомовними повідомленнями – це дія, яка відбулася та про яку повідомляють. Дія значною мірою визначає, яким чином структуровано решту об'єкта Intent, зокрема, що саме міститься в розділі даних і додаткових даних.

Для користування об'єктами `Intent` у межах свого додатка (або для використання іншими додатками, щоб викликати компоненти з вашого додатка) можна вказати власні дії. Зазвичай же слід використовувати константи дій, визначені класом `Intent` або іншими класами платформи. Ось кілька стандартних дій для запуску операції:

`ACTION_VIEW`. Використовують цю дію в об'єкті `Intent` із методом `startActivity()`, коли є певна інформація, яку операцію можна показати користувачеві, наприклад, фотографія в додатку галереї або адреса для перегляду в картографічному додатку;

`ACTION_SEND`. Його ще називають об'єктом Intent "share" (намір надати загальний доступ). Цю дію слід використовувати в об'єкті `Intent` із 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 за класом `Intent`. Інші дії визначають в інших частинах платформи Android. Наприклад, у `Settings` визначають дії, що відкривають ряд екранів програми для налаштування системи.

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

Якщо визначати власні дії, слід обов'язково використовувати як їхній префікс назви пакета додатка. Наприклад (рис. 2.77):

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

Рис. 2.77. **Установлення назви пакета під час визначення дії**

Дані

URI (об'єкт `Uri`), який посилається на дані, із якими буде виконуватися дія і/або тип MIME цих даних. Тип даних зазвичай визначено дією об'єкта `Intent`. Наприклад, якщо дією є `ACTION_EDIT`, у даних має міститися URI документа, який потрібно відредагувати.

Під час створення об'єкта Intent, крім URI, часто необхідно вказати тип даних (їхній тип MIME). Наприклад, операція, яка може виводити на екран зображення, швидше за все, не зможе відтворити аудіофайл, навіть якщо і в тих, і в інших даних будуть однакові формати URI. Тому вказівка типу MIME даних допомагає системі Android знайти найбільш прийнятний компонент для отримання об'єкта Intent. Однак тип MIME іноді можна успадкувати від URI, зокрема, коли дані становлять `content:` URI, який указує, що дані знаходяться на пристрої і ними управляє `ContentProvider`, а це дає можливість системі бачити тип MIME даних.

Щоб задати тільки URI – даних, слід викликати `setData()`. Щоб задати тільки тип MIME, викликають `setType()`. За потреби обидва цих параметри можна у явному вигляді задати за допомогою `setDataAndType()`.

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

Категорія

Рядок, що містить інші відомості про те, яким компонентом має виконуватися Оброблення цього об'єкта `Intent`. В об'єкт Intent можна помістити будь-яку кількість описів категорій, проте більшості об'єктів Intent категорії не потрібно. Ось 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`.

Указати категорію можна за допомогою `addCategory()`.

Наведені раніше властивості (назва компонента, дія, дані та категорія) становлять 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 наведено й іншу інформацію, яка не впливає на те, яким чином визначено необхідний компонент програми.

Об'єкт Intent також може містити:

Додаткові данні

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 із яких приймає два параметри: назва та значення ключа. Також можна створити об'єкт `Bundle` з усіма додатковими даними, потім уставити об'єкт `Bundle` в об'єкт `Intent` за допомогою методу `putExtras()`.

Наприклад, під час створення об'єкта Intent для надсилання листів із методом `ACTION_SEND` можна вказати отримувача за допомогою ключа `EXTRA_EMAIL`, а тему повідомлення – за допомогою ключа `EXTRA_SUBJECT`.

Клас `Intent` указує багато констант `EXTRA_*` для стандартних типів даних. Якщо потрібно оголосити власні додаткові ключі (для об'єктів Intent, які приймає додаток), обов'язково слід указати як префікс назву пакета додатка. Наприклад (рис. 2.78):

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

Рис. 2.78. **Встановлення назви пакета додатка**

Мітки

Мітки, визначені у класі `Intent`, які діють як метадані для об'єкта Intent. Мітки мають указувати системі Android, яким чином слід запускати

операцію (наприклад, до якої *завдання* має належати операція) і як з нею поводитися після запуску (наприклад, чи буде її вказано у списку останніх операцій).

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

Приклад явного об'єкта `Intent`

Явні об'єкти Intent використовують для запуску конкретних компонентів додатка, наприклад певної операції або служби. Щоб створити явний об'єкт Intent, слід задати назви компонента для об'єкта `Intent` – всі інші властивості об'єкта Intent можна не ставити.

Наприклад, якщо в додатку створили службу з назвою `DownloadService`, призначену для завантаження файлів з Інтернету, то для її запуску можна використовувати такий код (рис. 2.79):

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

Рис. 2.79. **Запуск служби DownloadService**

Конструктор `Intent(Context, Class)` надає `Context` із додатком, а компоненту – об'єкт `Class`. Фактично цей об'єкт `Intent` явно запускає клас `DownloadService` в додатку.

Приклад неявного об'єкта Intent

Неявний об'єкт Intent указує дію, якою може бути викликано будь-який наявний на пристрої додаток, здатний виконати цю дію. Неявні об'єкти Intent використовують, коли додаток не може виконати ту чи іншу дію, а інші додатки, швидше за все, можуть, і потрібно, щоб користувач мав можливість вибрати, яку програму для цього використовувати.

Наприклад, якщо є контент і потрібно, щоб користувач поділився ним з іншими людьми, слід створити об'єкт `Intent` із дією `ACTION_SEND` і додати додаткові дані, які вказують на контент, загальний доступ до якого слід надати. Коли за допомогою цього об'єкта `Intent` викликають `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, відправлений до методу `startActivity()`. У цьому випадку виклик закінчиться невдачею, а робота програми аварійно завершиться. Щоб перевірити, чи буде отримано операцією об'єкт `Intent`, слід викликати 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 по можливості слід відключити функцію, яка видає цей об'єкт `Intent` (рис. 2.80).

```
// 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);
}
```

Рис. 2.80. **Оброблення результату зі значенням null**

Примітка. У цьому випадку URI не використовують, а замість цього слід оголосити тип даних об'єкта Intent, щоб указати контент, що міститься в додаткових даних.

Під час 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 один прийнятний додаток, він буде одразу ж відкритим та отримає цей об'єкт Intent. Якщо об'єкт `Intent` приймає кілька операцій, система відображає діалогове вікно, у якому користувач може вибрати додаток для виконання цієї дії.

Примусове виконання блоку вибору додатка

Діалогове вікно вибору додатка зображено на рис. 2.81.

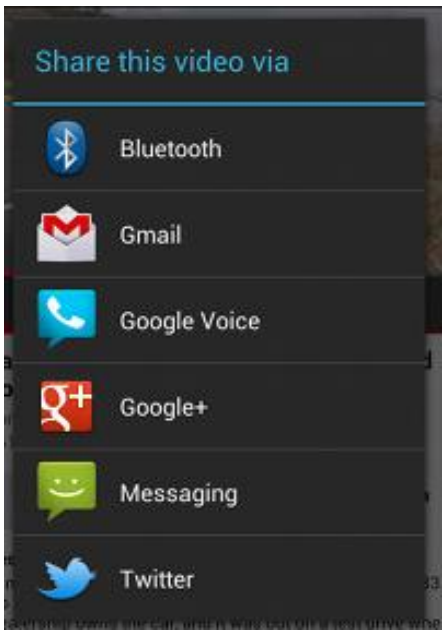

Рис. 2.81. **Діалогове вікно вибору**

За наявності декількох додатків, що реагують на неявний об'єкт Intent, користувач може вибрати потрібну програму та вказати, що вона буде за замовчуванням виконувати цю дію. Це зручно в разі дії, для виконання якої користувач хоче завжди використовувати один і той же додаток, наприклад, під час відкриття веб-сторінки (користувачі зазвичай використовують один і той же браузер).

Однак якщо на об'єкт `Intent` можуть відгукнутися кілька додатків, можливо користувач буде вважати за краще кожен раз використовувати іншу програму, тому слід явно виводити діалогове вікно вибору. У діалоговому вікні для вибору додатка користувачеві пропонується вибрати програму під час кожного запуску, вибрати, який додаток використовувати для дії (користувач не може вибрати програму, що використовується за замовчуванням). Наприклад, коли додаток виконує операцію *share* (поділитися) за допомогою дії `ACTION_SEND`, користувачі можуть, залежно від ситуації, робити це за допомогою різних додатків, тому слід завжди використовувати діалогове вікно вибору, як показано на рис. 2.80.

Щоб вивести на екран блок і вибрати програму, слід створити `Intent` за допомогою `createChooser()` і передати його до `startActivity()`, що подано у прикладі на рис. 2.82.

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

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

// Завжди використовуйте ресурси типу string для тексту інтерфейсу кори-
стувача.
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);
}
```

Рис. 2.82. **Дії для виведення на екран блоку з вибором програми**

2.7.3. Отримання неявного об'єкта Intent

Щоб указати, які неявні об'єкти Intent може приймати додаток, задати один або кілька фільтрів Intent для кожного компонента програми за допомогою елемента `intent-filter` у *файлі маніфесту*. Кожен фільтр Intent указує тип об'єктів Intent, які приймає компонент на підставі дії, даних і категорії, заданих в об'єкті Intent. Система передає неявний об'єкт Intent додатка, тільки якщо він може пройти через один із фільтрів Intent.

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

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

Кожен фільтр Intent визначено елементом `intent-filter` у файлі маніфесту додатка, зазначеному в оголошенні відповідного компонента програми (наприклад, в елементі `activity`). Усередині елемента `intent-filter` можна вказати тип об'єктів Intent, які будуть приймати, за допомогою одного або декількох із таких трьох елементів:

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

`data` – оголошує тип прийнятих даних, для чого використовано один або кілька атрибутів, що вказують різні складові частини URI-даних (`scheme`, `host`, `port`, `path` і т.д.) і тип MIME;

`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` до фільтра Intent. Методи `startActivity()` і `startActivityForResult()` обробляють усі об'єкти `Intent` так, як якщо б вони оголошували категорію `CATEGORY_DEFAULT`. Якщо не оголосити цю категорію у фільтрі Intent, ніякі неявні об'єкти Intent не буде передано в операцію.

У наступному прикладі оголошено операцію з фільтром Intent, що визначає отримання об'єкта Intent `ACTION_SEND`, коли дані належать до типу text (рис. 2.83):

```
<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>
```

Рис. 2.83. **Отримання об'єкта `Intent` `ACTION_SEND`, коли дані належать до типу `text`**

Можна створювати фільтри, у яких буде кілька примірників `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 категорій, необхідно створити кілька фільтрів `Intent`.

Обмеження доступу до компонентів

Використання фільтра Intent не є безпечним способом запобігання запуску 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 додатка за допомогою явного об'єкта Intent, якщо розробник визначить назви компонентів. Якщо важливо, щоб тільки власне додаток міг запускати один із компонентів, слід задати для атрибута `exported` цього компонента значення `false`.

Неявний об'єкт Intent перевіряють фільтром шляхом порівняння об'єкта Intent із кожним із цих трьох елементів. Щоб об'єкт Intent був доставлений компоненту, він має пройти всі три тести. Якщо він не буде відповідати хоча б одному з них, система Android не доставить цей об'єкт Intent компоненту. Однак, оскільки в компонента може бути кілька фільтрів Intent, об'єкт Intent, який не проходить через один із фільтрів компонента, може пройти через інший фільтр.

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

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

Приклади фільтрів

Щоб краще зрозуміти різні режими роботи фільтрів Intent, слід розглянути такий фрагмент із файла маніфесту додатка для роботи із соціальними мережами (рис. 2.84).

Перша операція `MainActivity` є основною точкою входу до додатка – це операція, яка відкривається, коли користувач запускає додаток натисканням на його значок:

дія `ACTION_MAIN` указує на те, що це основна точка входу, і не очікує ніяких даних об'єкта Intent;

категорія `CATEGORY_LAUNCHER` указує, що значок цієї операції слід помістити в засіб запуску додатків системи. Якщо елемент `activity` не містить указівок 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.

Другу операцію `ShareActivity` призначено 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 у `ShareActivity` безпосередньо з іншої програми, яка видає неявний об'єкт `Intent`, що відповідає одному із двох фільтрів `Intent`.

```
<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>
```

Рис. 2.84. **Фрагмент із файла маніфесту**

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

2.7.4. Використання очікувального об'єкта Intent

Об'єкт `PendingIntent` є оболонкою, у яку поміщено об'єкт `Intent`. Об'єкт `PendingIntent` в основному призначено для того, щоб надавати дозвіл зовнішнім додаткам на використання об'єкта `Intent`, що міститься в ньому так, якщо б він виконувався із процесу власного додатка.

Основні варіанти використання очікувального об'єкта `Intent`:

оголошення об'єкта Intent, який має буде виконано, коли користувач виконує дію з *повідомленням* (`NotificationManager` системи Android виконує `Intent`);

оголошення об'єкта Intent, який має буде виконувано, коли користувач виконує дію з віджетом додатка (додаток головного екрана виконує `Intent`);

оголошення об'єкта Intent, який має буде виконано в зазначений час у майбутньому (`AlarmManager` системи Android виконує `Intent`).

Оскільки кожен об'єкт `Intent` призначено для оброблення компонентом програми, який належать до певного типу (`Activity`, `Service` або `BroadcastReceiver`), об'єкт `PendingIntent` також слід створювати з урахуванням цієї обставини. Під час використання об'єкта Intent, що очікує, програма не буде виконувати об'єкт Intent викликом, наприклад, `startActivity()`. Замість цього необхідно буде оголосити необхідний тип компонента під час створення `PendingIntent` шляхом виклику відповідного методу для створення:

метод `PendingIntent.getActivity()` для `Intent`, який запускає `Activity`;

метод `PendingIntent.getService()` для `Intent`, який запускає `Service`;

метод `PendingIntent.getBroadcast()` для `Intent`, який запускає `BroadcastReceiver`.

Якщо тільки додаток *не приймає очікувальні* об'єкти Intent від інших додатків, зазначені раніше методи створення `PendingIntent` є єдиними методами `PendingIntent`, які 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 очікувальних об'єктів Intent наведено в документації по кожному з відповідних варіантів використання, наприклад, в інструкціях, присвячених API-інтерфейсам: *Повідомлення* і *Віджети додатків*.

2.7.5. Дозвіл об'єктів Intent

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

дія об'єкта Intent;

дані об'єкта Intent (тип URI і даних);

категорія об'єкта Intent.

У наступних підрозділах описано, яким чином об'єкти Intent зіставляються з відповідними компонентами, а саме, яким має бути фільтр Intent, оголошений у файлі маніфесту додатка.

Тестування дії

Для задавання приймаючих дій об'єкту Intent фільтр може оголошувати будь-яку (у тому числі нульову) кількість елементів `action`. Наприклад (рис. 2.85):

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

Рис. 2.85. **Оголошення елементів у Intent фільтрі для задавання дій**

Щоб пройти через цей фільтр, дія, указана в об'єкті `Intent`, має відповідати одній або декільком вимогам, переліченим у фільтрі.

Якщо у фільтрі не перелічено будь-які дії, об'єкту Intent не буде чому відповідати, 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'єкту Intent фільтр Intent може оголошувати будь-яку (у тому числі нульову) кількість елементів `category`. Наприклад (рис. 2.86):

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

Рис. 2.86. **Оголошення елементів у Intent фільтрі для встановлення прийнятих категорій**

Щоб об'єкт Intent пройшов тестування категорії, усі категорії, наведені в об'єкті `Intent`, мають відповідати категорії з фільтра. Зворотне не потрібно – фільтр Intent може оголошувати й інші категорії, яких немає в об'єкті `Intent`, об'єкт `Intent` все одно пройде тест. Тому об'єкт Intent без категорій завжди пройде цей тест, незалежно від того, які категорії оголошено в фільтрі.

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

Тестування даних

Для зазначення вхідних даних об'єкту Intent фільтр Intent може оголошувати будь-яку (у тому числі нульову) кількість елементів `data`. Наприклад (рис. 2.87):

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

Рис. 2.87. **Оголошення елементів у `Intent` фільтрі для встановлення вхідних даних**

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`.

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

Кожен із цих атрибутів є необов'язковим в елементі `data`, проте має 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, указаний в об'єкті Intent, порівнюється з URI з фільтру, порівнювання виконується тільки з тими складовими частинами URI, які наведено у фільтрі. Наприклад:

якщо у фільтрі вказана тільки схема, то всі URI до цієї схеми будуть відповідати фільтру;

якщо у фільтрі вказано схему та повноваження, але відсутній шлях, усі URI з такими ж схемою та повноваженнями пройдуть фільтр, а їхні шляхи враховувати не будуть;

якщо у фільтрі вказано схему, повноваження та шлях, то тільки URI з такими ж схемою, повноваженнями та шляхом пройдуть фільтр.

Примітка. Шлях може бути зазначено з підстановлювальним символом (*), щоб була необхідна тільки часткова відповідність назві шляху.

Під час виконання тестування даних порівнюється і URI, і тип MIME, зазначені в об'єкті Intent, з URI і типом MIME з фільтра. Діють такі правила:

а) 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;

б) об'єкт Intent, у 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;

в) об'єкт Intent, у якому є тип MIME, але відсутній URI, пройде цей тест, тільки якщо у фільтрі вказано той же тип MIME і не вказано формат URI;

г) об'єкт Intent, у якому є і URI, і тип MIME (явний чи той, який можна вивести з URI), пройде тільки частину цього тесту, яка перевіряє тип MIME, у тому випадку, якщо цей тип збігається з типом, наведеним у фільтрі. Він пройде 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. Інакше кажучи, передбачено, що компонент підтримує дані `content:` і `file:`, якщо у його фільтрі вказано тільки тип MIME.

Це останнє правило (правило (г)) відображає очікування того, що компоненти будуть у змозі отримувати локальні дані з файла або від постачальника контенту. Тому їхні фільтри можуть містити тільки тип даних, а явно вказувати схеми `content:` і `file:` не потрібно. Це типовий випадок. Наприклад, такий елемент `data`, як наведено далі, повідомляє системі Android, що компонент може отримувати дані зображень від постачальника контенту та виводити їх на екран (рис. 2.88):

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

Рис. 2.88. **Приклад використання елемента data для отримання зображень**

Оскільки наявні дані, переважно, поширюються постачальниками контенту, фільтри, у яких зазначено тип даних і немає URI, імовірно, є найпоширенішими.

Іншою стандартною конфігурацією є фільтри зі схемою та типом даних. Наприклад, такий елемент `data`, який наведено далі, повідомляє системі Android, що компонент може отримувати відеодані з мережі для виконання дії (рис. 2.89):

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

Рис. 2.89. **Приклад використання елемента data для отримання відеоданих**

Зіставлення об'єктів Intent

Об'єкти Intent зіставляють із 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 із фільтрами Intent, у яких зазначено дію `ACTION_MAIN` і категорію `CATEGORY_LAUNCHER`.

Програма може використовувати зіставлення об'єктів Intent таким же чином. У `PackageManager` є набір методів `query...()`, які повертають усі компоненти, здатні прийняти певний об'єкт, а також схожий набір методів `resolve ... ()`, які визначають найбільш прийнятний компонент, здатний реагувати на об'єкт `Intent`. Наприклад, метод `queryIntentActivities()` повертає переданий як аргумент список усіх 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 перелічують ті з них, які можуть відгукнутися. Є схожий метод для приймачів широкомовних повідомлень (`{@link android.content.pm.PackageManager # queryBroadcastReceivers`).

2.8. Випливні повідомлення

Випливне повідомлення забезпечує простий зворотний зв'язок про операцію в невеликому випливному вікні. Воно займає лише той обсяг місця, який необхідний для повідомлення, і поточна діяльність залишається видимою та інтерактивною. Наприклад, вихід із пошти до відправлення листа викликає випливне вікно "Чернетку збережено", щоб повідомити, що можна продовжити редагування пізніше. Випливні повідомлення зникають автоматично після закінчення часу очікування (рис. 2.90).

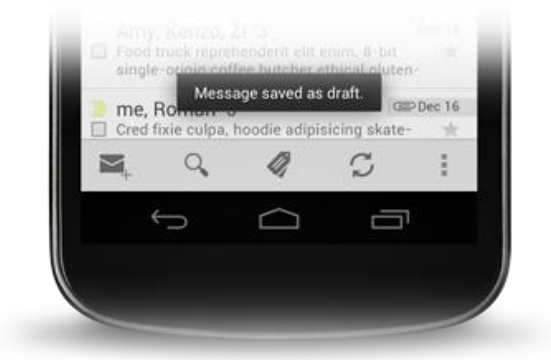

Рис. 2.90. **Випливні повідомлення**

Якщо потрібна відповідь користувача на повідомлення про стан системи, слід розглянути використання `Notification`.

2.8.1. Основи

По-перше, слід створити екземпляр об'єкта `Toast` за допомогою одного з методів `makeText()`. Цей метод приймає три параметри: `Context` додатка, текстове повідомлення та тривалість випливного повідомлення.

Він повертає правильно ініціалізований об'єкт Toast. Можна відобразити випливне повідомлення за допомогою `show()`, як показано на рис. 2.91.

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

Рис. 2.91. **Відображення випливного повідомлення за допомогою `show()`**

Цей приклад демонструє все, що потрібно для більшості випливних повідомлень. Рідко може знадобитися щось ще. Однак можна захотіти по-іншому позиціонувати випливне повідомлення, або навіть використовувати власний макет, замість простого текстового повідомлення. У наступних розділах описано, як можна це зробити.

Можна також прив'язати свої методи та уникнути прив'язування до об'єкта Toast, подібно до такого (рис. 2.92):

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

Рис. 2.92. **Прив'язка та об'єкт `Toast`**

2.8.2. Позиціонування випливного повідомлення

Стандартне випливне повідомлення з'являється в нижній частині екрана, по центру по горизонталі. Можна змінити це положення 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 вирішити, що випливне повідомлення має з'являтися у верхньому лівому кутку, можна встановити тяжіння так (рис. 2.93):

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

Рис. 2.93. **Встановлення тяжіння**

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

Створення користувацького уявлення випливного повідомлення.

Якщо простого текстового повідомлення недостатньо, можна створити власний макет для свого випливного повідомлення. Для цього визначити макет View у XML або в коді додатка та передати кореневий об'єкт `View` методу `setView (View)`.

Наприклад, можна створити макет для випливного повідомлення, показаного на скриншоті справа за допомогою такого XML (збережіть як layout / custom_toast.xml) (рис. 2.94):

```
<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>
```

Рис. 2.94. **Створення макету для випливного повідомлення**

Зверніть увагу, що `ID` елемента `LinearLayout` становить `custom_toast_container`. Мають використовувати цей `ID` та `ID` файла макета XML `custom_toast`, щоб наповнити макет, як показано на рис. 2.95.

```
LayoutInflater inflater = getLayoutInflater();
View layout = inflater.inflate(R.layout.custom_toast,
                (ViewGroup) findViewById(R.id.custom_toast_container));
```

Рис. 2.95. **Використання `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();
```

Закінчення рис. 2.95

Спочатку слід отримати `LayoutInflater` за допомогою `getLayoutInflater()` (або `getSystemService()`), а потім розгорнути макет із XML, використовуючи `inflate (int, ViewGroup)`. Перший параметр – це ID ресурсу макета, а другий – це кореневий View. Можна використовувати цей макет, щоб знайти більше об'єктів View у макеті, щоб зібрати та визначити вміст для елементів ImageView і TextView. Нарешті, створити нове випливне повідомлення за допомогою `Toast(Context)` та встановити такі властивості випливного повідомлення, як тяжіння та тривалість. Потім викликати `setView(View)` і передати йому наповнений макет. Тепер можна відобразити випливне повідомлення за допомогою призначеного для користувача макета викликом методу `show()`.

Примітка. Не слід використовувати загальнодоступний конструктор для випливного повідомлення, якщо не потрібно визначати макет за допомогою `setView(View)`. Якщо немає користувацького макета, мають використовувати `makeText(Context, int, int)` для створення випливного повідомлення.

Запитання для самодіагностики

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 виконують Оброблення події кнопки *Back*?

Розділ 3. Служби та сервіси мобільних платформ

Мета: надати основні принципи служби та сервісу мобільних платформ.

Компетентності: проектувати та розробляти мобільний додаток з використанням мережних сервісів.

Знати: основні типи мережних сервісів мобільної платформи та засоби їх використання; API мобільної платформи, що дозволяє використовувати компонент стільникового зв'язку, оброблення телефонних викликів, мобільний інтернет та геолокацію.

Вміти: програмувати компонент стільникового зв'язку; обробляти телефонні виклики, мобільний інтернет та геолокацію; ефективно формувати комунікаційну стратегію під час роботи в команді.

Нести відповідальність за якість реалізації функціональних вимог до мобільного додатка та строків його реалізації.

Основні питання:

1. Програмний компонент: Служби (Services).
2. Засоби запуску служб.
3. Управління життєвим циклом служби.
4. Створення прив'язаних груп.

Ключові слова: служба, програмний компонент, процес, фоновий потік, маніфест, життєвий цикл.

3.1. Програмний компонент: Служби (Services)

`Service` є компонентом програми, який може виконувати тривалі операції у фоновому режимі й не містить інтерфейс користувача. Інший компонент програми може запустити службу, яка продовжить роботу у фоновому режимі навіть у тому випадку, коли користувач перейде в інший додаток. Крім того, компонент може прив'язатися до служби для взаємодії з нею та навіть виконувати міжпроцесну взаємодію (IPC). Наприклад, служба може обробляти мережні транзакції, відтворювати музику, виконувати введення-виведення файла або взаємодіяти з постачальником контенту – і все це у фоновому режимі.

Фактично служба може приймати дві форми:

- **запущена.** Служба є "запущеною", коли компонент додатка (наприклад, операція) запускає її викликом `startService()`. Після запуску служба може працювати у фоновому режимі протягом необмеженого часу, навіть якщо знищений компонент, який її запустив. Зазвичай запущена служба виконує одну операцію й не повертає результатів викликаючому компоненту. Наприклад, вона може завантажувати файл по мережі. Коли операцію виконано, служба має зупинитися самостійно;
- **прив'язана.** Служба є "прив'язаною", коли компонент програми прив'язується до неї викликом `bindService()`. Прив'язана служба пропонує інтерфейс клієнт-сервер, який дозволяє компонентам взаємодіяти зі службою, відправляти запити, отримувати результати та навіть робити це між різними процесами за допомогою міжпроцесної взаємодії (IPC). Прив'язана служба працює тільки доти, поки до неї прив'язаний інший компонент програми. До служби може бути прив'язано кілька функцій одночасно, але коли вони скасовують прив'язування, служба знищується.

Хоча в цій документації ці два типи служб обговорюються окремо, служба може працювати обома способами: вона може бути занедбаною (і працювати протягом необмеженого часу) і допускати прив'язування. Це залежить від реалізації пари методів зворотного виклику: `onStartCommand()` дозволяє запускати служби, а `onBind()` дозволяє виконувати прив'язування.

Незалежно від стану програми (запущеної, прив'язаної або обох відразу), будь-який компонент програми може використовувати службу (навіть з окремого додатка), подібно до того, як будь-який компонент може використовувати операцію, запустивши її з допомогою `Intent`. Однак можна оголосити закриту службу у файлі маніфесту та заблокувати доступ до неї з інших додатків.

Увага! Служба працює в основному потоці головного процесу, вона не створює свого потоку та не виконується в окремому процесі (якщо не вказано інше). Це означає, що, якщо служба збирається виконувати будь-яку роботу з високим навантаженням ЦП або блокувальні операції (наприклад, відтворення MP3 або мережеві операції), мають створити у службі новий потік для виконання цієї роботи. Використовуючи окремий потік, знижують ризик виникнення помилок "Додаток не відповідає", і основний потік програми може відпрацьовувати взаємодію користувача з операціями.

3.1.1. Основи

Щоб створити службу, необхідно створити підклас класу `Service` (або одного з наявних його підкласів). У реалізації необхідно визначити деякі методи зворотного виклику, які обробляють ключові моменти життєвого циклу служби та за потреби надають механізм прив'язування компонентів. Найбільш важливі методи зворотного виклику, які необхідно визначити, це:

`onStartCommand()`. Система викликає цей метод, коли інший компонент, наприклад, операція, запитує запуск цієї служби, викликаючи `startService()`. Після виконання цього методу служба запускається та може протягом необмеженого часу працювати у фоновому режимі. Якщо реалізувати такий метод, то треба зупинити службу за допомогою виклику `stopSelf()` або `stopService()` (якщо потрібно тільки забезпечити прив'язування, реалізовувати цей метод не обов'язково);

`onBind()`. Система викликає цей метод, коли інший компонент хоче виконати прив'язування до служби (наприклад, для виконання віддаленого виклику процедури) шляхом виклику `bindService()`. У реалізації цього методу мають забезпечити інтерфейс, який клієнти використовують для взаємодії зі службою, повертаючи `IBinder`. Завжди необхідно реалізовувати цей метод, але якщо не дозволено прив'язування, необхідно повертати значення `null`;

`onCreate()`. Система викликає цей метод під час першого створення служби для виконання одноразових процедур налаштування (перед викликом `onStartCommand()` або `onBind()`). Якщо службу вже запущено, цей метод не викликається;

`onDestroy()`. Система викликає цей метод, коли служба більше не використовується та виконується її знищення. Служба має реалізувати це для очищення таких ресурсів, як: потоки, зареєстровані приймачі, ресивери та ін. Це останній виклик, який отримує служба.

Якщо компонент запускає службу за допомогою виклику `startService()` (що призводить до виклику `onStartCommand()`), то служба продовжує роботу, поки вона не зупиниться самостійно за допомогою `stopSelf()` або інший компонент не зупинить її за допомогою виклику `stopService()`.

Якщо компонент викликає `bindService()` для створення служби (і `onStartCommand()` не викликається), то служба працює, поки до неї прив'язано компонент. Як тільки виконується скасування прив'язування служби до всіх клієнтів, система знищує службу.

Система Android буде примусово зупиняти службу тільки в тому випадку, коли не вистачає пам'яті та необхідно відновити системні операції, які відображаються на передньому плані. Якщо службу прив'язано до операції, яка відображається на передньому плані, менш імовірно, що її буде знищено, і якщо службу оголошено для виконання у фоновому режимі (як обговорювалося раніше), вона майже ніколи не буде знищуватися. В іншому випадку, якщо службу було запущено і вона є тривалою, система з часом буде опускати її положення у списку фонових завдань, і служба стане дуже чутливою до знищення. Якщо служба працює, то мають передбачити витончене оброблення її перезапуску системою. Якщо система знищує службу, вона запускає її, як тільки знову з'являється доступ до ресурсів (хоча це також залежить від значення, що повертається методом `onStartCommand()`, як обговорюється далі).

У наступних розділах описано способи створення служб кожного типу та використання їх з інших компонентів додатка.

3.1.2. Що краще: служба чи потік?

Служба – це просто компонент, який може виконуватися у фоновому режимі, навіть коли користувач не взаємодіє з додатком. Отже, мають створювати службу тільки в тому випадку, якщо вам потрібно саме це.

Якщо вам потрібно виконати роботу за межами основного потоку, але тільки тоді, коли користувач взаємодіє з додатком, то вам, імовірно, слід створити новий потік, а не службу. Наприклад, якщо потрібно відтворювати певну музику, але тільки під час роботи, операції, можна створити потік в `onCreate()`, запустити його виконання в методі `onStart()`, а потім зупинити його в методі `onStop()`. Також слід розглянути можливість використання класу `AsyncTask` або `HandlerThread`, замість звичайного класу `Thread`. У документі *Процеси і потоки* міститься додаткова інформація про ці потоки.

Слід пам'ятати, що, якщо використовувати службу, вона виконується в основному потоці додатка за замовчуванням, тому мають створити новий потік у службі, якщо вона виконує інтенсивні або блокувальні операції.

Оголошення служби в маніфесті

Усі служби, як і операції (та інші компоненти), має бути оголошено у файлі маніфесту додатка.

Щоб оголосити службу, слід додати елемент `<service>` як дочірній елемент `<application>`. Наприклад (рис. 3.1):

```
<manifest ... >
  ...
  <application ... >
    <service android:name=".ExampleService" />
    ...
  </application>
</manifest>
```

Рис. 3.1. **Оголошення служби**

Додаткові відомості про оголошення служби в маніфесті дивіться у довідці про елемент `<service>`.

Є інші атрибути, які можна включити в елемент `<service>` для задавання властивостей, наприклад необхідних для запуску дозволів, і процесу, у якому має виконуватися служба. Атрибут `android:name` є єдиним обов'язковим атрибутом: він указує назву класу для служби. Після публікації додатка не слід змінювати цю назву, оскільки це може зруйнувати код із-за залежності від явних намірів, використовуваних, щоб запустити або прив'язати службу (ознайомтеся з публікацією "Речі, які не можна змінювати у блозі розробників").

Для забезпечення безпеки додатка **завжди слід використовувати явний намір під час запуску або прив'язування** `Service` і не розказувати фільтрів намірів для служби. Якщо важливо допустити деяку невизначеність щодо того, яка служба запускається, можна надати фільтри для ваших намірів служб і виключити назву компонента з `Intent`, але потім мають встановити пакет для наміру за допомогою `setPackage()`, який забезпечує достатнє усунення неоднозначності для цільової служби.

Додатково можна забезпечити доступність служби тільки для цього додатка, включивши атрибут `android:exported` і встановивши для нього значення `false`. Це не дозволяє іншим програмам запускати службу навіть під час використання явного наміру.

3.1.3. Створення запущеної служби

Запущена служба це служба, яку запускає інший компонент викликом `startService()`, що призводить до виклику методу `onStartCommand()` служби.

Під час запуску служба має термін життя, що не залежить від компонента, який її запустив, і може працювати у фоновому режимі протягом необмеженого часу, навіть якщо знищено компонент, який її запустив. Тому після виконання своєї роботи служба має зупинитися самостійно за допомогою виклику методу `stopSelf()` або її може зупинити інший компонент за допомогою виклику методу `stopService()`.

Компонент програми, наприклад операція, може запустити службу, викликавши метод `startService()` і передавши об'єкт `Intent`, який указує службу та будь-які дані, які служба має використовувати. Служба отримує цей об'єкт `Intent` у методі `onStartCommand()`.

Слід припустити, що операції потрібно зберегти певні дані в мережевій базі даних. Операція може запустити службу та надати їй дані для збереження, передавши намір методу `startService()`. Служба отримує намір у методі `onStartCommand()`, підключається до Інтернету та виконує транзакцію з базою даних. Коли транзакцію виконано, служба зупиняється самостійно та знищується.

Увага! За замовчуванням служби працюють у тому ж процесі, що й додаток, у якому їх оголошено, а також в основному потоці цього додатка. Тому, якщо ваша служба виконує інтенсивні або блокувальні операції, тоді як користувач взаємодіє з операцією з того ж додатка, служба буде сповільнювати виконання операції. Щоб уникнути негативного впливу на швидкість роботи програми, мають запустити новий потік усередині служби.

Традиційно є два класи, які можна успадкувати для створення запущеної служби:

`Service`. Це базовий клас для всіх служб. Коли наслідують цей клас, важливо створити новий потік, у якому буде виконуватися вся робота служби, оскільки за замовчуванням служба використовує основний потік додатка, що може уповільнити будь-яку операцію, яку виконує цей додаток;

`IntentService`. Цей підклас класу `Service`, який використовує робочий потік для оброблення всіх запитів запуску по черзі. Це оптимальний варіант, якщо не потрібно, щоб служба обробляла декілька запитів одночасно. Достатньо реалізувати метод `onHandleIntent()`, який отримує намір для кожного запиту запуску, дозволяючи виконувати фонову роботу.

У наступних підрозділах описано, як реалізувати службу з допомогою будь-якого із цих класів.

Спадкування класу IntentService

Оскільки більшості запущених додатків не потрібно обробляти декілька запитів одночасно (що може бути дійсно небезпечним сценарієм), імовірно, буде краще, якщо реалізувати свою службу з допомогою класу `IntentService`.

Клас `IntentService` робить таке:

створює робочий потік за замовчуванням, який виконує всі наміри, доставлені в метод `onStartCommand()`, окремо від основного потоку додатка;

створює робочу чергу, яка передає наміри по одному в реалізацію методу `onHandleIntent()`, тому не мають турбуватися щодо багатопотоковості;

зупиняє службу після оброблення всіх запитів запуску, тому ніколи не потрібно викликати `stopSelf()`;

надає реалізацію методу `onBind()`, яка повертає `null`;

надає реалізацію методу `onStartCommand()` за замовчуванням, яка відправляє намір у робочу чергу та потім у реалізацію `onHandleIntent()`.

Усе це означає, що достатньо реалізувати метод `onHandleIntent()` для виконання роботи, наданої клієнтом (Хоча, крім того, мають надати маленький конструктор для служби).

Усе, що потрібно: конструктор і реалізація класу `onHandleIntent()`.

Якщо вирішено визначити також і такі інші методи зворотного виклику, як: `onCreate()`, `onStartCommand()` або `onDestroy()`, обов'язково слід викликати реалізацію суперкласу, щоб клас `IntentService` міг правильно обробляти життєвий цикл робочого потоку.

На рис. 3.2 наведено приклад реалізації класу `IntentService`.

```
public class HelloIntentService extends IntentService {
  /**
* Потрібен конструктор, він має викликати IntentService(String)
* конструктор з назвою для робочого потоку.
   */
  public HelloIntentService() {
      super("HelloIntentService");
  }
```

Рис. 3.2. **Реалізації класу `IntentService`**

```
  /**
* IntentService викликає цей метод із робочого потоку за замовчуванням.
* Коли цей метод виконався, IntentService
* зупиняє службу, за необхідності.
   */
  @Override
  protected void onHandleIntent(Intent intent) {
    /* Як правило, ми будемо виконувати деяку роботу тут, наприклад,
скачувати файл. Для нашого прикладу, ми просто зупиняємо виконання на 5
секунд. */
    long endTime = System.currentTimeMillis() + 5*1000;
    while (System.currentTimeMillis() < endTime) {
      synchronized (this) {
        try {
          wait(endTime - System.currentTimeMillis());
        } catch (Exception e) {

        }
      }
    }
  }
}
```

Закінчення рис. 3.2

Наприклад, метод `onStartCommand()` має повертати реалізацію за замовчуванням (яка доставляє намір `onHandleIntent()`) (рис. 3.3).

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Toast.makeText(this, "service starting",
                   Toast.LENGTH_SHORT).show();
    return super.onStartCommand(intent,flags,startId);
}
```

Рис. 3.3. **Реалізація методу `onStartCommand()`**

Крім `onHandleIntent()`, єдиний метод, із якого не потрібно викликати суперклас, це метод `onBind()` (але його потрібно реалізовувати тільки у випадку, якщо ваша служба допускає прив'язування).

У наступному підрозділі показано, як реалізовано службу такого ж типу під час спадкування базового класу `Service`, яка містить набагато більше коду, але може підійти, якщо потрібно обробляти одночасні запити запуску.

Спадкування класу Service

Як показано в попередньому підрозділі, використання класу `IntentService` значно спрощує реалізацію запущеної служби. Проте, якщо необхідно, щоб ваша служба підтримувала багатопотоковість (замість оброблення запитів запуску через робочу чергу), можна успадковувати клас `Service` для оброблення кожного наміру.

Як приклад наведено таку реалізацію класу `Service`, яка виконує ту ж роботу, як і у прикладі раніше та використовує клас `IntentService`. Тобто для кожного запиту запуску він використовує робочий потік для виконання завдання та обробляє запити по-одному (рис. 3.4).

```
public class HelloService extends Service {
  private Looper mServiceLooper;
  private ServiceHandler mServiceHandler;

  // Обробник, який отримує повідомлення з потоку
  private final class ServiceHandler extends Handler {
    public ServiceHandler(Looper looper) {
      super(looper);
    }

    @Override
    public void handleMessage(Message msg) {
      /* Як правило, ми будемо виконувати деяку роботу тут, наприклад,
скачувати файл. Для нашого прикладу ми просто зупиняємо виконання на 5
секунд */
    long endTime = System.currentTimeMillis() + 5*1000;
    while (System.currentTimeMillis() < endTime) {
      synchronized (this) {
        try {
          wait(endTime - System.currentTimeMillis());
        } catch (Exception e) {
        }
      }
    }
```

Рис. 3.4. **Реалізація класу Service**

```
    /* Зупиняємо сервіс за допомогою startId, таким чином ми не зупиняє-
мо сервіс у середині оброблення іншої роботи */
    stopSelf(msg.arg1);
  } // END ServiceHandler

  @Override
  public void onCreate() {

// Запуск потоку, у якому виконується служба. Зверніть увагу, що ми
// створюємо окремий потік, тому що служба зазвичай працює в
// основному потоці процесу, який ми не хочемо заблокувати. Ми
// також задали йому фоновий пріоритет, так що напружена робота не пору-
шить нашого інтерфейсу користувача.

    HandlerThread thread = new HandlerThread("ServiceStartArguments",
                                    Process.THREAD_PRIORITY_BACKGROUND)
;
    thread.start();
    // Отримуємо HandlerThread і використовувати його для нашого оброб-
ника */
    mServiceLooper = thread.getLooper();
    mServiceHandler = new ServiceHandler(mServiceLooper);
  }

  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();

    /* Для кожного запускаємо запит, відправляємо повідомлення, щоб по-
чати роботу, і відправляємо ID, так що ми знаємо, який запит ми зупини-
ли, коли закінчили роботу */
    Message msg = mServiceHandler.obtainMessage();
    msg.arg1 = startId;
    mServiceHandler.sendMessage(msg);
    return START_STICKY;
  }

  @Override
  public IBinder onBind(Intent intent) {
    // Ми не надаємо прив'язування, тому повертаємо null
    return null;
  }
```

Продовження рис. 3.4

```
  @Override
  public void onDestroy() {
    Toast.makeText(this, "service done", Toast.LENGTH_SHORT).show();
  }
}
```

Закінчення рис. 3.4

Як можна бачити, цей код значно довший, ніж код із використанням класу `IntentService`.

Однак, оскільки обробляють кожен виклик `onStartCommand()` самостійно, можна виконувати декілька запитів одночасно. Цей код виконує не зовсім таку роботу, але за потреби можна створювати нові потоки для кожного запиту та відразу запускати їх (а не чекати завершення попереднього запиту).

Зверніть увагу, що метод `onStartCommand()` має повертати ціле число. Це ціле число описує, як система має продовжувати виконання служби в разі, коли система знищила її (як описано раніше, реалізація за замовчуванням для класу `IntentService` обробляє цю ситуацію). Значення, що повертається методом `onStartCommand()`, має бути однією з таких констант:

`START_NOT_STICKY`. Якщо система знищує службу після повернення з `onStartCommand()`, *не потрібно* повторно створювати службу, якщо немає очікуючих доставки намірів. Це найбезпечніший варіант, що дозволяє уникнути запуску вашої служби, коли це не потрібно та додаток може просто перезапустити будь які незавершені завдання;

`START_STICKY`. Якщо система знищує службу після повернення з `onStartCommand()`, слід створити службу та викликати `onStartCommand()`, але не передавати останнім намір повторно. Замість цього система викликає метод `onStartCommand()` із наміром, що має значення `null`, якщо немає очікувальних намірів для запуску служби. Якщо очікувальні наміри є, їх доставляють. Це підходить для мультимедійних програвачів (або подібних служб), які не виконують команди, а працюють незалежно та чекають завдання;

`START_REDELIVER_INTENT`. Якщо система знищує службу після повернення з `onStartCommand()`, слід повторно створити службу та викликати `onStartCommand()` з останнім наміром, що було доставлено у службу.

Усі очікувальні наміри доставляють по черзі. Це підходить для служб, що активно виконують завдання, яке має бути відновлено негайно, наприклад для завантаження файла.

Для отримання додаткових відомостей про значення дивіться довідкову документацію за посиланням для кожної константи.

Запуск служби

Можна запустити службу операції або іншого компонента, передавши об'єкт `Intent` (указує службу, яку потрібно запустити) у `startService()`. Система Android викликає метод `onStartCommand()` служби та передає їй `Intent` (ні в якому разі не слід викликати метод `onStartCommand()` напряму).

Наприклад, операція може запустити службу із прикладу в попередньому розділі (`HelloSevice`), використовуючи явний намір за допомогою `startService()` (рис. 3.5):

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

Рис. 3.5. **Реалізація `startService()`**

Метод `startService()` повертається негайно, і система Android викликає метод служби `onStartCommand()`. Якщо служба ще не виконується, система спочатку викликає `onCreate()`, а потім `onStartCommand()`.

Якщо служба також не становить прив'язування, намір, що доставляється з допомогою `startService()`, є єдиним режимом зв'язку між компонентом програми та службою. Однак, якщо потрібно, щоб служба відправляла результат назад, клієнт, який запускає службу, може створити об'єкт `PendingIntent` для повідомлення (із допомогою `getBroadcast()`) і доставити його на службу в об'єкті `Intent`, який запускає службу. Потім служба може використовувати повідомлення для доставлення результату.

Кілька запитів запуску служби приводять до кількох відповідних викликів методу `onStartCommand()` служби. Однак для її зупинки достатньо тільки одного запиту на зупинку служби (із допомогою `stopSelf()` або `stopService()`).

Зупинка служби

Запущена служба має управляти своїм життєвим циклом. Тобто, система не зупиняє та не знищує службу, якщо не потрібно відновити

пам'ять системи, і служба продовжує роботу після повернення з методу `onStartCommand()`. Тому служба має зупинятися самостійно за допомогою виклику методу `stopSelf()` або інший компонент може зупинити її за допомогою виклику методу `stopService()`.

Отримавши запит на зупинку з допомогою `stopSelf()` або `stopService()`, система якомога швидше знищує службу.

Однак, якщо служба обробляє кілька запитів `onStartCommand()` одночасно, не можна зупиняти службу після завершення оброблення запиту запуску, оскільки, імовірно, уже отримано новий запит запуску (зупинка в кінці першого запиту призвела б до переривання другого). Щоб уникнути цієї проблеми, можна використовувати метод `stopSelf(int)`, який гарантує, що запит на зупинку служби завжди засновано на самому останньому запиті запуску. Тобто, коли викликають `stopSelf(int)`, передають ідентифікатор запиту запуску (ідентифікатор `startId`, доставлений в `onStartCommand()`), якому відповідає запит зупинки. Тоді, якщо служба отримає новий запит запуску до того, як зможуть викликати `stopSelf(int)`, вона не буде збігатися та службу не буде зупинено.

Увага! Додаток обов'язково має зупиняти свої служби після закінчення роботи, щоб уникнути витрачання ресурсів і споживання енергії акумулятора. За потреби інші компоненти можуть зупинити службу з допомогою виклику методу `stopService()`. Навіть якщо можна виконувати прив'язування служби, слід завжди зупиняти службу самостійно, якщо вона коли-небудь отримала виклик `onStartCommand()`.

3.1.4. Створення прив'язаної служби

Прив'язана служба це служба, яка допускає прив'язування до неї компонентів програми за допомогою виклику `bindService()` для створення довготривалого з'єднання (і зазвичай не дозволяє компонентам запускати її з допомогою виклику `startService()`).

Мають створити прив'язану службу, коли потрібно взаємодіяти зі службою операцій та інших компонентів додатка чи показувати деякі функції цього додатка іншим за допомогою міжпроцесної взаємодії (IPC).

Щоб створити прив'язану службу, необхідно реалізувати метод зворотного виклику `onBind()` для повернення об'єкта `IBinder`, який визначає інтерфейс взаємодії зі службою. Після цього інші компоненти програми можуть викликати метод `bindService()` для вилучення інтерфейсу

та почати викликати методи служби. Служба існує тільки для обслуговування прив'язаного до неї компонента, тому, коли немає компонентів, прив'язаних до служби, система знищує її (не потрібно зупиняти прив'язану службу, як це потрібно для служби, запущеної за допомогою `onStartCommand()`).

Щоб створити прив'язану службу, необхідно, у першу чергу, визначити інтерфейс взаємодії клієнта зі службою. Цей інтерфейс між службою та клієнтом має бути реалізацією об'єкта `IBinder`, яку служба повинна повертати з методу зворотного виклику `onBind()`. Після того як клієнт отримує об'єкт `IBinder`, він може почати взаємодію зі службою за допомогою цього інтерфейсу.

Одночасно до служби може бути прив'язано кілька клієнтів. Коли клієнт закінчує взаємодію зі службою, він викликає `unbindService()` для скасування прив'язування. Як тільки не залишається ні одного клієнта, прив'язаного до служби, система знищує службу.

Існує декілька способів реалізації прив'язаної служби, і ці реалізації більш складні, ніж реалізації запущеної служби, тому обговорення прив'язаної служби наведено в окремому розділі "Прив'язані служби".

3.1.5. Відправлення повідомлень користувачеві

Після запуску служба може повідомляти користувача про події, використовуючи *Випливні повідомлення* або *Повідомлення в рядку стану*.

Випливне повідомлення це повідомлення, що короткочасно з'являється на поверхні поточного вікна, тоді як повідомлення в рядку стану це значок у рядку стану з повідомленням, що користувач може вибрати, щоб виконати дію (таку як запуск операції).

Зазвичай повідомлення в рядку стану є найбільш зручним рішенням, коли завершується якась фонова робота (наприклад завершено завантаження файла) і користувач може діяти. Коли користувач вибирає повідомлення в розширеному вигляді, повідомлення може запустити операцію (наприклад, для перегляду завантаженого файла).

3.1.6. Запуск служби на передньому плані

Служба переднього плану це служба, із якою користувач активно обізнаний, і тому вона не є кандидатом для видалення системою в разі нестачі пам'яті. Служба переднього плану має виводити повідомлення в рядок стану, що знаходиться під заголовком "Постійні". Це означає,

що повідомлення не може бути видалено, поки службу не буде зупинено або видалено з переднього плану.

Наприклад, музичний програвач, який відтворює музику зі служби, має бути налаштовано на роботу на передньому плані, оскільки користувач точно знає про його роботу. Повідомлення в рядку стану може показувати поточний твір і дозволяти користувачу запускати операцію для взаємодії з музичним програвачем.

Для запиту на виконання служби на передньому плані слід викликати метод `startForeground()`. Цей метод має два значення: ціле число, яке однозначно ідентифікує повідомлення, та об'єкт `Notification` для рядка стану. Наприклад (рис. 3.6):

```
Notification notification = new Notification(R.drawable.icon,
                                        getText(R.string.ticker_text),
                                         System.currentTimeMillis());
Intent notificationIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingIntent =
                        PendingIntent.getActivity(this,
                                                  0,
                                                  notificationIntent,
                                                  0);
notification.setLatestEventInfo(this,
                                getText(R.string.notification_title),
                                getText(R.string.notification_message),
                                pendingIntent);
startForeground(ONGOING_NOTIFICATION_ID, notification);
```

Рис. 3.6. **Реалізація методу** `startForeground()`

Увага! Цілочисельний ідентифікатор ID, який передають у метод `startForeground()`, не має дорівнювати `0`.

Щоб видалити службу з переднього плану, слід викликати `stopForeground()`. Цей метод містить логічне значення, яке вказує, чи потрібно видаляти повідомлення в рядку стану. Цей метод не зупиняє службу. Однак, якщо слід зупинити службу, що працює на передньому плані, повідомлення буде видалено.

3.1.7. Управління життєвим циклом служби

Життєвий цикл служби набагато простіший, ніж життєвий цикл операції. Однак більш важливо приділити пильну увагу тому, як ця служба створюється та знищується, оскільки служба може працювати у фоновому режимі без відома користувача.

Життєвий цикл служби від створення до знищення може відбуватися двома різними шляхами:

- **запущена служба.** Служба створюється, коли інший компонент викликає метод `startService()`. Потім служба працює протягом необмеженого часу та має зупинитися самостійно за допомогою виклику методу `stopSelf()`. Інший компонент також може зупинити службу за допомогою виклику методу `stopService()`. Коли служба зупиняється, система знищує її;
- **прив'язана служба.** Служба створюється, коли інший компонент (клієнт) викликає метод `bindService()`. Потім клієнт взаємодіє зі службою через інтерфейс `IBinder`. Клієнт може закрити з'єднання за допомогою виклику методу `unbindService()`. До однієї служби може бути прив'язано кілька клієнтів, і коли вони скасовують прив'язування, система знищує службу (служба не має зупинятися самостійно).

Ці два способи необов'язково працюють незалежно один від одного. Тобто можна прив'язати службу, яку вже було запущено за допомогою методу `startService()`. Наприклад, фонову музичну службу може бути запущено за допомогою виклику методу `startService()` з об'єктом `Intent`, який ідентифікує музику для відтворення. Пізніше, наприклад, коли користувач хоче отримати доступ до управління програвачем, операція може встановити прив'язування до служби за допомогою виклику методу `bindService()`. У подібних випадках методи `stopService()` і `stopSelf()` фактично не зупиняють службу, поки не буде скасовано прив'язування всіх клієнтів.

3.1.8. Реалізація зворотних викликів життєвого циклу

Подібно до операції, служба містить методи зворотного виклику життєвого циклу, які можна реалізувати для контролю за змінами стану служби та виконання роботи у відповідні моменти часу. Зазначена далі базова служба показує кожний із методів життєвого циклу (рис. 3.7; 3.8).

```
public class ExampleService extends Service {
  int mStartMode;       // визначаємо що робити, якщо сервіс не працює
  IBinder mBinder;      // інтерфейс для клієнтів
  boolean mAllowRebind; // визначаємо, чи слід використовувати onRebind
  @Override
  public void onCreate() {
    // Сервіс створено
  }
  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    // сервіс запускається
    return mStartMode;
  }
  @Override
  public IBinder onBind(Intent intent) {
    //Клієнт прив'язується до сервісу за допомогою bindService()
    return mBinder;
  }
    @Override
    public boolean onUnbind(Intent intent) {
    // Всі клієнти непов'язані з unbindService()
    return mAllowRebind;
  }
  @Override
  public void onRebind(Intent intent) {
    // Клієнт прив'язується до сервісу за допомогою bindService()
    // після onUnbind(), що вже був викликаний
  }
  @Override
  public void onDestroy() {
    // Послуга більше не використовується і знищується
  }
}
```

Рис. 3.7. **Реалізація базової службу, що показує кожний із методів життєвого циклу**

Примітка. На відміну від методів зворотного виклику життєвого циклу операції, не потрібно викликати реалізацію суперкласу цих методів зворотного виклику.

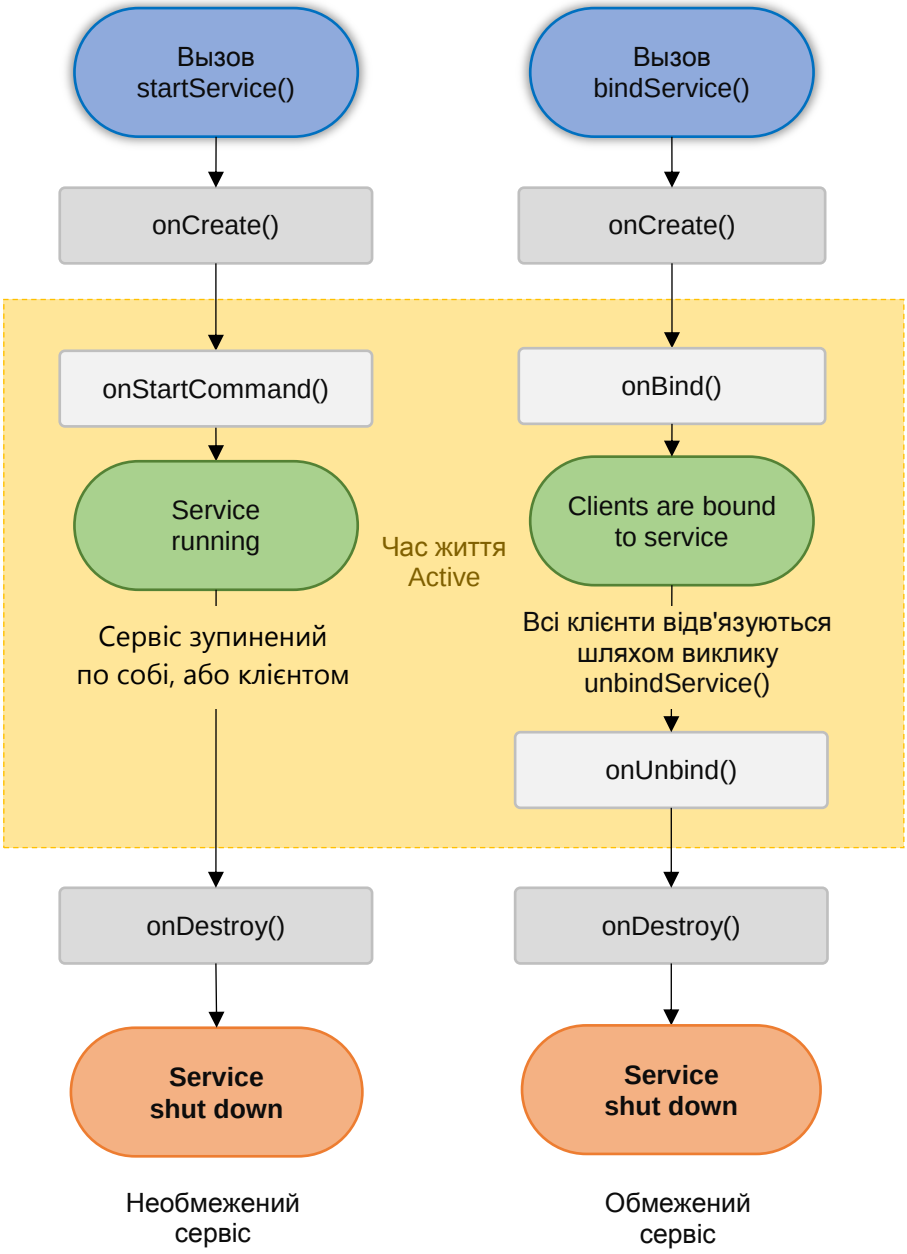

Рис. 3.8. **Життєвий цикл служби**

На схемі зліва показано життєвий цикл, коли службу створено за допомогою методу `StartService()`, а на схемі справа життєвий цикл, коли службу створено за допомогою методу `bindService()`.

За допомогою реалізації цих методів можна відстежувати два вкладених цикли в життєвому циклі служби:

• **увесь життєвий цикл** служби відбувається між викликом методу `onCreate()` і поверненням із методу `onDestroy()`. Подібно до операції,

служба виконує початкове налаштування в методі `onCreate()` і звільняє всі ресурси, що залишилися в методі `onDestroy()`. Наприклад, служба відтворення музики може створити потік для відтворення музики в методі `onCreate()`, потім зупинити потік у методі `onDestroy()`.

Методи `onCreate()` і `onDestroy()` викликають для всіх служб, незалежно від методу створення: `startService()` чи `bindService()`;

• **активний життєвий цикл** служби починається з виклику методу `onStartCommand()` чи `onBind()`. Кожен метод спрямовується наміром `Intent`, який був переданий методу `startService()` чи `bindService()`, відповідно.

Якщо службу запущено, активний життєвий цикл закінчується одночасно із закінченням усього життєвого циклу (служба активна навіть після повернення з методу `onStartCommand()`). Якщо служба є прив'язаною, активний життєвий цикл закінчується, коли повертається метод `onUnbind()`.

Примітка. Хоча запущена служба зупиняється за допомогою виклику методу `stopSelf()` чи `stopService()`, для служби не існує відповідного зворотного виклику (немає зворотного виклику `onStop()`). Тому, якщо служба не прив'язана до клієнта, система знищує її під час зупинки служби – метод `onDestroy()` є єдиним отримуваним методом зворотного виклику.

Рис. 3.8 ілюструє типові методи зворотного виклику для служби. Хоча на рисунку відокремлені служби, створені за допомогою методу `startService()`, від служб, створених за допомогою методу `bindService()`, слід пам'ятати, що будь-яка служба, незалежно від способу запуску, дозволяє клієнтам виконувати прив'язування до неї. Тому служба, спочатку створена за допомогою методу `onStartCommand()` (клієнтом, який викликав `startService()`), може отримувати виклик методу `onBind()` (коли клієнт викликає метод `bindService()`).

3.1.9. Прив'язані служби (Bound Services)

Прив'язана служба надає інтерфейс типу "клієнт-сервер". Прив'язана служба дозволяє компонентам додатка (наприклад операціям) взаємодіяти зі службою, відправляти запити, отримувати результати, навіть робити те ж саме з іншими процесами через IPC. Прив'язана служба, зазвичай, працює, поки інший компонент додатка прив'язаний до неї. Вона не працює постійно у фоновому режимі.

Прив'язана служба становить реалізацію класу `Service`, яка дозволяє іншим програмам прив'язуватися до нього і взаємодіяти з ним. Щоб забезпечити прив'язування служби, спочатку необхідно реалізувати метод зворотного виклику `onBind()`. Цей метод повертає об'єкт `IBinder`.

Він визначає програмний інтерфейс, за допомогою якого клієнти можуть взаємодіяти зі службою.

Прив'язування до запущеної службі

Можна створити службу, яка одночасно є і запущеною, і прив'язаною. Це означає, що службу можна запустити шляхом виклику методу `startService()`, який дозволяє службі працювати необмежений час, а також дозволяє клієнтам прив'язуватися до неї за допомогою виклику методу `bindService()`.

Якщо дозволити запуск і прив'язування служби, то після її запуску система *не* знищує її після скасування всіх прив'язувань клієнтів. Замість цього необхідно явно зупинити службу, викликавши метод `stopSelf()` чи `stopService()`.

Незважаючи на те що зазвичай необхідно реалізовувати або метод `onBind()`, або метод `onStartCommand()`, у деяких випадках потрібно реалізувати обидва ці методи. Наприклад, у музичному програвачі може виявитися корисним дозволити виконання служби протягом необмеженого часу, а також забезпечити її прив'язування. Таким чином, операція може запустити службу для відтворення музики, яка буде відтворюватися навіть після виходу користувача із програми. Після його повернення до додатка операція може скасувати прив'язування до служби, щоб повернути управління відтворенням.

Для прив'язування до служби клієнт може викликати метод `bindService()`. Після прив'язування він має надати реалізацію методу `ServiceConnection`, який слугує для відстеження підключення до служби. Метод `bindService()` повертається негайно без значення, проте, коли система Android установлює підключення "клієнт-служба", вона викликає метод `onServiceConnected()` для `ServiceConnection`, щоб видати об'єкт `IBinder`, який клієнт може використовувати для взаємодії зі службою.

Одночасно до служби можуть підключитися відразу кілька клієнтів. Однак система викликає метод `onBind()` служби для отримання об'єкта `IBinder` тільки під час першого прив'язування клієнта. Після чого система видає такий же об'єкт `IBinder` для будь-яких додаткових клієнтів, які виконують прив'язування, без повторного виклику методу `onBind()`.

Коли скасовується прив'язування останнього клієнта від служби, система знищує службу (якщо тільки службу не було так само запущено методом `startService()`).

Найважливішу роль в реалізації прив'язаної служби відіграє визначення інтерфейсу, який повертає метод зворотного виклику `onBind()`. Існує кілька різних способів визначення інтерфейсу `IBinder` служби. Кожен із них розглянуто далі в наступному підрозділі.

Створення прив'язаної служби. Під час створення служби, що забезпечує прив'язування, потрібен об'єкт `IBinder`, що забезпечує програмний інтерфейс, за допомогою якого клієнти можуть взаємодіяти зі службою. Існує три способи визначення такого інтерфейсу:

Розширення класу `Binder`.

Якщо служба є приватною і надається в межах власного додатка, а також виконується в тому ж процесі, що й клієнт (загальний процес), створювати інтерфейс слід шляхом розширення класу `Binder` і повернення його екземпляра з методу `onBind()`. Клієнт отримує об'єкт `Binder`, після чого він може використовувати його для отримання прямого доступу до загальнодоступних методів, наявного або в реалізації `Binder`, або навіть у `Service`.

Цей спосіб є кращим, коли служба просто виконується у фоновому режимі для додатка. Цей спосіб не підходить для створення інтерфейсу тільки тоді, коли власна служба використовується іншими додатками або в окремих процесах.

Використання об'єкта `Messenger`.

Якщо необхідно, щоб інтерфейс служби був доступний для різних процесів, його можна створити за допомогою об'єкта `Messenger`. Таким чином, служба визначає об'єкт `Handler`, відповідний різним типам об'єктів `Message`. Цей об'єкт `Handler` є основою для об'єкта `Messenger`, який, у свою чергу, надає клієнтові об'єкт `IBinder`, завдяки чому останній може відправляти команди у службу за допомогою об'єктів `Message`. Крім того, клієнт може визначити об'єкт `Messenger` для самого себе, що дозволяє службі повертати повідомлення клієнта.

Це найпростіший спосіб організувати взаємодію процесів, оскільки `Messenger` організовує чергу всіх запитів у межах одного потоку, тому не потрібно робити свою службу потокобезпечною.

Використання мови AIDL.

AIDL (Android Interface Definition Language) виконує всю роботу з розподілу об'єктів на примітиви, які операційна система може розпізнати та розподілити за процесами для організації взаємодії між ними (IPC).

Попередній спосіб із використанням об'єкта `Messenger` фактично засновано на AIDL, оскільки це його базова структура. Як уже згадувалося раніше, об'єкт `Messenger` створює чергу з усіх запитів клієнтів у межах одного потоку, тому служба одночасно отримує тільки один запит. Однак, якщо необхідно, щоб служба обробляла одночасно відразу кілька запитів, можна використовувати AIDL безпосередньо. У такому випадку служба має підтримувати багатопотоковість і повинна бути потокобезпечною.

Щоб використовувати AIDL безпосередньо, необхідно створити файл `.aidl`, який визначає програмний інтерфейс. Цей файл використовує інструменти SDK Android для створення абстрактного класу, який реалізує інтерфейс, і забезпечує взаємодію процесів і який у подальшому можна розширити у службі.

Примітка. У більшості додатків **не слід** використовувати AIDL для створення та прив'язування служби, оскільки для цього може знадобитися підтримка багатопотоковості, що, у свою чергу, може призвести до більш складної реалізації.

Розширення класу Binder

Якщо служба використовує тільки локальний додаток і не взаємодіє з різними процесами, можна реалізувати власний клас `Binder`, за допомогою якого клієнт отримує прямий доступ до загальнодоступних методів у службі.

Примітка. Цей варіант підходить тільки в тому випадку, якщо клієнт і служба виконуються всередині однієї програми та процесу, що є найбільш поширеною ситуацією. Наприклад, розширення класу відмінно підійде для музичного додатка, у якому необхідно прив'язати операцію до власної служби додатка, яка відтворює музику у фоновому режимі.

Ось як це зробити:

1. Створити у службі екземпляр класу `Binder` із такими характеристиками:

екземпляр містить загальнодоступні методи, які може викликати клієнт;

екземпляр повертає поточний екземпляр класу `Service`, що містить загальнодоступні методи, які може викликати клієнт;

екземпляр повертає екземпляр іншого класу, розміщений у службі, що містить загальнодоступні методи, які може викликати клієнт.

2. Повернути цей екземпляр класу `Binder` із методу зворотного виклику `onBind()`.

3. Клієнт має отримати клас `Binder` від методу зворотного виклику `onServiceConnected()` і виконати виклики до прив'язаної служби за допомогою наданих методів.

Примітка. Служба та клієнт мають виконуватися в одному й тому ж додатку, оскільки в цьому випадку клієнт може транслювати повернутий об'єкт і належним чином викликати його API-інтерфейси. Крім того, служба та клієнт мають виконуватися в межах одного і того ж процесу, оскільки цей спосіб не має на увазі будь-якого розподілу за процесами.

Далі наведено приклад служби, яка надає клієнтам доступ до методів за допомогою реалізації класу `Binder` (рис. 3.9):

```
public class LocalService extends Service {
    // Binder надається клієнтам
    private final IBinder mBinder = new LocalBinder();
    // Генератор випадкових чисел
    private final Random mGenerator = new Random();

    /* Клас, який використовують для клієнта Binder. Тому що ми знаємо,
що ця послуга завжди працює в тому ж процесі, як її клієнти, нам
не потрібно мати справу з IPC. */
    public class LocalBinder extends Binder {
      LocalService getService() {
         // Поверніть цей екземпляр LocalService, так що клієнти можуть
викликати публічні методи
          return LocalService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }

    /** метод для клієнтів */
    public int getRandomNumber() {
      return mGenerator.nextInt(100);
    }
}
```

Рис. 3.9. **Служба, яка надає клієнтам доступ до методів за допомогою реалізації класу `Binder`**

Об'єкт `LocalBinder` надає для клієнтів метод `getService()`, щоб вони могли отримати поточний екземпляр класу `LocalService`. Завдяки цьому клієнти можуть викликати загальнодоступні методи у службі. Наприклад, клієнти можуть викликати метод `getRandomNumber()` зі служби.

Далі наведено приклад операції, яка виконує прив'язування до класу `LocalService` і викликає метод `getRandomNumber()` під час натискання кнопки (рис. 3.10):

```
public class BindingActivity extends Activity {
    LocalService mService;
    boolean mBound = false;

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

    @Override
    protected void onStart() {
      super.onStart();
      // Прив'язування до LocalService
      Intent intent = new Intent(this, LocalService.class);
      bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
    }

    @Override
    protected void onStop() {
      super.onStop();
      // Відв'язування від служби
      if (mBound) {
        unbindService(mConnection);
        mBound = false;
      }
    }

    /* Викликається під час натискання на кнопку (кнопка в файлі макета
надає цей метод за допомогою атрибута android:onClick) */
```

Рис. 3.10. **Операція, яка виконує прив'язування до класу `LocalService` і викликає метод `getRandomNumber()` під час натискання кнопки**

```
    public void onButtonClick(View v) {
      if (mBound) {
        // Виклик методу з LocalService.
        // Однак, якщо цей виклик був таким, що може викликати зависання,
то цей запит має відбуватися в окремому потоці, щоб уникнути уповільнення
роботи дії. */
        int num = mService.getRandomNumber();
        Toast.makeText(this, "number: " + num, Toast.LENGTH_SHORT)
             .show();
      }
    }

    /* Визначає функцію зворотного виклику для зв'язування служби, пере-
дається bindService() */
    private ServiceConnection mConnection = new ServiceConnection() {
      @Override
      public void onServiceConnected(ComponentName className,
                                     IBinder service) {
        // Ми зобов'язані LocalService, кинути IBinder і отримати примірник
LocalService
        LocalBinder binder = (LocalBinder) service;
        mService = binder.getService();
        mBound = true;
      }

      @Override
      public void onServiceDisconnected(ComponentName arg0) {
        mBound = false;
      }
    };
}
```

Закінчення рис. 3.10

У наведеному раніше прикладі показано, як клієнт прив'язується до служби за допомогою реалізації `ServiceConnection` і зворотного виклику `onServiceConnected()`.

Примітка. У наведеному прикладі не виконано явне скасування прив'язування до служби, проте всім клієнтам слід скасовувати прив'язування у відповідних термінах (наприклад, коли операція припиняється).

Приклади коду наведено у статтях, присвячених класам `LocalService.java` й `LocalServiceActivities.java`, в `ApiDemos`.

Використання об'єкта Messenger

Якщо необхідно, щоб служба взаємодіяла з віддаленими процесами, для надання інтерфейсу служби можна скористатися об'єктом `Messenger`. Такий підхід дозволяє організувати взаємодію між процесами (IPC) без необхідності використовувати AIDL.

Ось короткий огляд того, як слід використовувати об'єкт `Messenger`:

служба реалізує об'єкт `Handler`, який отримує зворотний виклик для кожного виклику від клієнта;

об'єкт `Handler` використовують для створення об'єкта `Messenger` (який є посиланням на об'єкт `Handler`);

об'єкт `Messenger` створює об'єкт `IBinder`, який служба повертає клієнтам із методу `onBind()`;

клієнти використовують отриманий об'єкт `IBinder` для створення екземпляра об'єкта `Messenger` (який посилається на об'єкт `Handler` служби), що використовується клієнтом для відправлення об'єктів `Message` у службу;

служба отримує кожен об'єкт `Message` у своєму об'єкті `Handler`, зокрема у методі `handleMessage()`.

Таким чином, для клієнта відсутні "методи" для відправлення виклику служби. Замість цього клієнт відправляє "повідомлення" (об'єкти `Message`), які служба отримує у своєму об'єкті `Handler`.

Далі наведено приклад служби, яка використовує інтерфейс `Messenger` (рис. 3.11):

```
public class MessengerService extends Service {
  /** Команда до служби, щоб відобразити повідомлення */
  static final int MSG_SAY_HELLO = 1;

  /** Обробник вхідних повідомлень від клієнтів. */
  class IncomingHandler extends Handler {
    @Override
    public void handleMessage(Message msg) {
      switch (msg.what) {
        case MSG_SAY_HELLO:
          Toast.makeText(getApplicationContext(),
```

Рис. 3.11. **Реалізація інтерфейсу** `Messenger`

```
                        "hello!",
                        Toast.LENGTH_SHORT).show();
          break;
        default:
          super.handleMessage(msg);
      }
    }
  }

  /** Цільові ми публікуємо для клієнтів для відправлення повідомлень
IncomingHandler. */
  final Messenger mMessenger = new Messenger(new IncomingHandler());

  /** Під час прив'язування до служби, ми повертаємо інтерфейс до нашого
посланця для відправлення повідомлення у службу. */
  @Override
  public IBinder onBind(Intent intent) {
    Toast.makeText(getApplicationContext(),
                   "binding",
                   Toast.LENGTH_SHORT).show();
    return mMessenger.getBinder();
  }
}
```

Закінчення рис. 3.11

Зверніть увагу, що метод `handleMessage()` в об'єкті `Handler` – це місце, де служба отримує вхідні об'єкти `Message` та вирішує, що робити далі, керуючись елементом `what`.

Клієнту потрібно лише створити об'єкт `Messenger` на основі об'єкта `IBinder`, повернутого службою, і відправити повідомлення за допомогою методу `send()`. Далі наведено приклад того, як проста операція виконує прив'язування до служби та відправляє їй повідомлення `MSG_SAY_HELLO` (рис. 3.12):

```
public class ActivityMessenger extends Activity {
  /** Комунікатор для спілкування зі службою. */
  Messenger mService = null;
```

Рис. 3.12. **Операція виконує прив'язування до служби та відправляє їй повідомлення MSG_SAY_HELLO**

```
  /** Прапорець, який указує, чи визвали ми прив'язування на службу. */
  boolean mBound;

  /** Клас для взаємодії з основним інтерфейсом сервісу.  */
  private ServiceConnection mConnection = new ServiceConnection() {
    public void onServiceConnected(ComponentName className,
                                   IBinder service) {
      /* Це викликається, коли з'єднання з сервісом було установлено,
що дає нам об'єкт, який ми можемо використовувати для взаємодії зі служ-
бою. Ми спілкуємося зі службою за допомогою Messenger, тому тут ми отри-
муємо на боці клієнта уявлення про це з вихідного об'єкта IBinder. */
      mService = new Messenger(service);
      mBound = true;
    }

    public void onServiceDisconnected(ComponentName className) {
      // Це викликається, коли з'єднання з сервісом було несподівано
вимкнено, тобто, процес його впав.
      mService = null;
      mBound = false;
    }
  };

    public void sayHello(View v) {
      if (!mBound) return;
      // Створити та відправити повідомлення у службу, використовуючи
підтримуване 'what' значення
      Message msg = Message.obtain(null, MessengerService.MSG_SAY_HELLO,
                                   0, 0);
      try {
        mService.send(msg);
      } catch (RemoteException e) {
        e.printStackTrace();
      }
    }

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

Продовження рис. 3.12

```
        setContentView(R.layout.main);
    }

    @Override
    protected void onStart() {
      super.onStart();
      // Прив'язка до служби
      bindService(new Intent(this, MessengerService.class),
                  mConnection, Context.BIND_AUTO_CREATE);
    }

    @Override
    protected void onStop() {
        super.onStop();
        // Відв'язування від служби
        if (mBound) {
            unbindService(mConnection);
            mBound = false;
        }
    }
}
```

Закінчення рис. 3.12

Зверніть увагу, що в цьому прикладі не показано, як служба відповідає клієнту. Якщо потрібно, щоб служба реагувала, необхідно створити об'єкт `Messenger` і у клієнті. Потім, коли клієнт отримує зворотний виклик `onServiceConnected()`, вона відправляє у службу об'єкт `Message`, який включає об'єкт `Messenger` клієнта як значення параметра `replyTo` методу `send()`.

Приклад організації двостороннього обміну повідомленнями наведено у прикладах коду `MessengerService.java` (*служба*) и `MessengerService Activities.java` (*клієнт*).

Порівняння з AIDL. Коли необхідно організувати взаємодію між процесами використання об'єкта `Messenger` для інтерфейсу – набагато простішим є реалізація за допомогою AIDL, оскільки об'єкт `Messenger` поміщає в чергу всі запити до служби, тоді як інтерфейс, заснований виключно на AIDL, відправляє у службу кілька запитів одночасно, для чого потрібна підтримка багатопотоковості.

У більшості додатків служба не повинна підтримувати багатопотоковість, тому під час використання об'єкта `Messenger` служба може одночасно обробляти один запит. Якщо важливо, щоб служба була багатопотоковою, для визначення інтерфейсу слід використовувати `AIDL`.

Прив'язування до служби

Для прив'язування до служби компоненти додатка (клієнти) можуть використовувати метод `bindService()`. Після цього система Android викликає метод `onBind()` служби, який повертає об'єкт `IBinder` для взаємодії зі службою.

Прив'язування виконується асинхронно. `bindService()` повертається відразу ж і не повертає клієнту об'єкт `IBinder`. Для отримання об'єкта `IBinder` клієнту необхідно створити екземпляр `ServiceConnection` і передати його в метод `bindService()`. Інтерфейс `ServiceConnection` включає метод зворотного виклику, який система використовує для того, щоб видати об'єкт `IBinder`.

Примітка. Виконати прив'язування до служби можуть тільки операції, інші служби та постачальники контенту. **Не можна** самостійно виконати прив'язування до служби з ресивера.

Тому для прив'язування до служби із клієнта необхідно виконати такі дії:

1. Реалізувати інтерфейс `ServiceConnection`.

Ваша реалізація має перевизначати такі два методи зворотного виклику:

`onServiceConnected()` – система викликає цей метод, щоб видати об'єкт `IBinder`, повернутий методом `onBind()` служби;

`onServiceDisconnected()` – система Android викликає цей метод у разі непередбаченої втрати з'єднання зі службою, наприклад під час перебоїв у роботі служби або її завершення. Цей метод не викликається, коли клієнт скасовує прив'язування.

2. Викликати метод `bindService()`, передавши в нього реалізацію інтерфейсу.

3. Коли система викликає метод зворотного виклику `onService Connected()`, можна приступити до виконання викликів до служби за допомогою методів, визначених інтерфейсом.

4. Щоб відключитися від служби, викликати метод `unbindService()`.

У разі знищення клієнта виконується скасування його прив'язування до служби, проте завжди слід відміняти прив'язування після завершення взаємодії зі службою або в разі припинення операції, щоб служба могла завершити свою роботу, коли вона не використовується.

Далі наведено приклад фрагмента коду для підключення клієнта до створеної раніше служби шляхом *розширення класу* `Binder` – клієнту потрібно лише передати повернутий об'єкт `IBinder` у клас `LocalService` та запросити екземпляр `LocalService` (рис. 3.13):

```
LocalService mService;
private ServiceConnection mConnection = new ServiceConnection() {
  // Викликається, коли з'єднання із сервісом установлено
  public void onServiceConnected(ComponentName className,
                                 IBinder service) {
    /* Тому що ми зобов'язані явному сервісу, який працює в нашому влас-
ному процесі, ми можемо відкинути його IBinder до конкретного класу та
отримати доступ до нього. */
    LocalBinder binder = (LocalBinder) service;
    mService = binder.getService();
    mBound = true;
  }

  // Викликається, коли з'єднання із сервісом роз'єднується несподівано
  public void onServiceDisconnected(ComponentName className) {
    Log.e(TAG, "onServiceDisconnected");
    mBound = false;
  }
};
```

Рис. 3.13. **Розширення класу `Binder`**

За допомогою цього інтерфейсу `ServiceConnection` клієнт може виконати прив'язування до служби, передавши її в методі `bindService()`.

Наприклад (рис. 3.14):

```
Intent intent = new Intent(this, LocalService.class);
bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
```

Рис. 3.14. **Реалізування методу `bindService()`**

Перший параметр у методі `bindService()` становить об'єкт `Intent`, який явно називає службу для прив'язування (хоча перехід може бути і неявним).

Другий параметр – це об'єкт `ServiceConnection`.

Третій параметр становить прапорець, який указує параметри прив'язування. Зазвичай ними є `BIND_AUTO_CREATE`, що створює служба, якщо вона вже не виконується. Інші можливі значення: `BIND_DEBUG_UNBIND` і `BIND_NOT_FOREGROUND` або `0`, якщо значення відсутнє.

Управління життєвим циклом прив'язаної служби

Коли виконується скасування прив'язування служби до всіх клієнтів, система Android знищує таку службу (якщо її не було запущено разом з `onStartCommand()`). У такому випадку не потрібно управляти життєвим циклом своєї служби, якщо вона є виключно прив'язаною службою: система Android управляє нею на підставі прив'язування служби до будь-яких інших клієнтів.

Однак, якщо потрібно реалізувати метод зворотного виклику `onStartCommand()`, необхідно явно зупинити службу, оскільки в цьому випадку вона вважається запущеною. У такому випадку служба виконується до тих пір, поки сама не зупинить свою роботу за допомогою методу `stopSelf()` або до тих пір, поки інший компонент не викличе метод `stopService()`, незалежно від прив'язування служби до будь-яких клієнтів.

Крім того, якщо служба запущена та приймає прив'язування, то під час виклику системою методу `onUnbind()` можна повернути `true`, якщо потрібно отримати виклик до `onRebind()` за наступного прив'язування до служби (замість отримання виклику до методу `onBind()`). Метод `onRebind()` повертає значення `void`, однак клієнт як і раніше отримує об'єкт `IBinder` у своєму методі зворотного виклику `onServiceConnected()`. На рис. 3.15 проілюстровано логіку життєвого циклу такого роду:

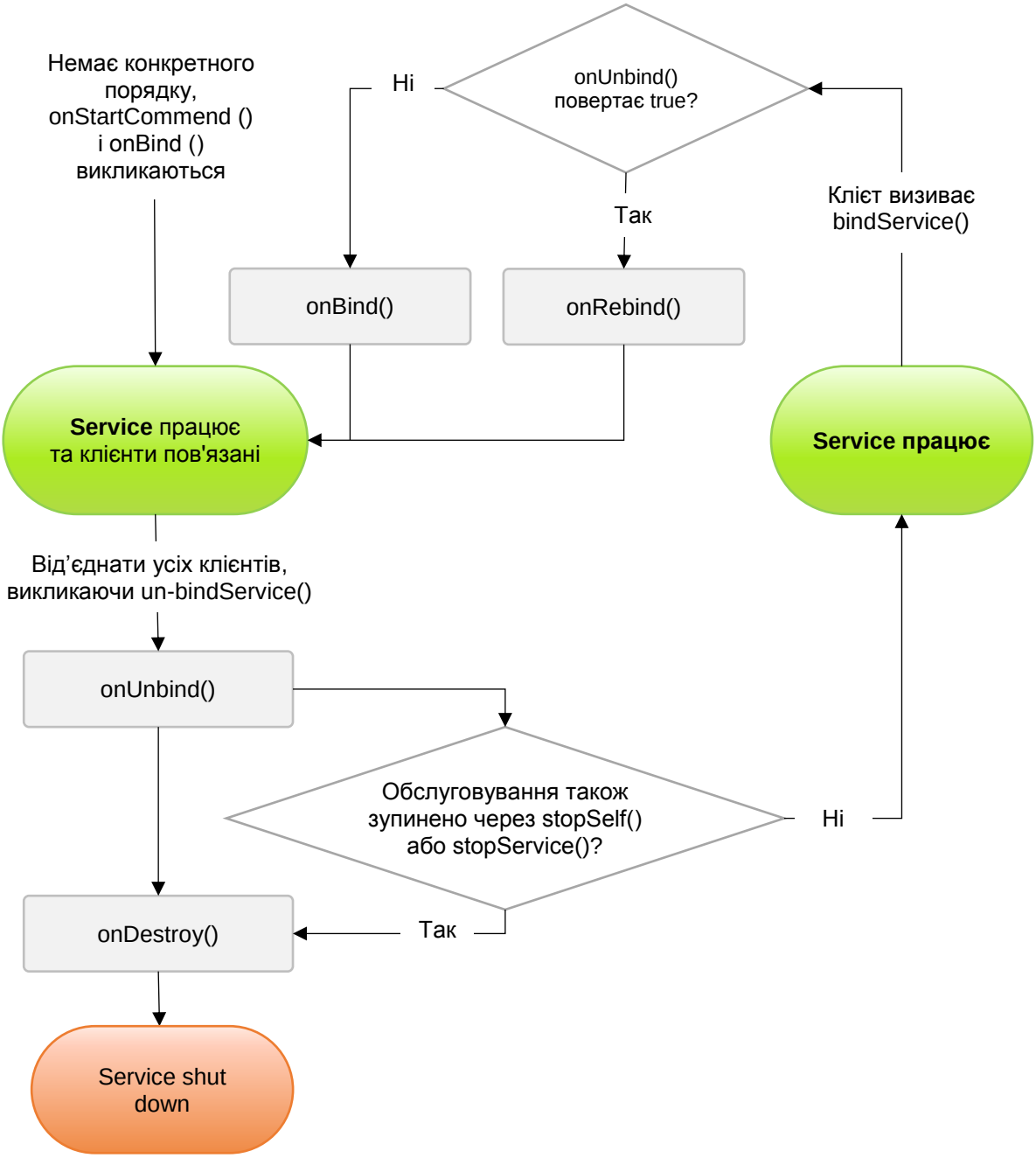

Рис. 3.15. **Життєвий цикл запущеної служби, для якої виконується прив'язування**

Запитання для самодіагностики

1. У чому полягає призначення служб?
2. Наведіть життєвий цикл служби.
3. Опишіть процес створення служби.
4. Як запускати службу на передньому плані процесів?
5. Опішить процес прив'язування до служби.

Розділ 4. Збереження та оброблення даних у мобільних додатках

Мета: надати основні принципи щодо збереження та оброблення даних у мобільних додатках.

Компетентності: проектувати та розробляти мобільний додаток з локальним обробленням даних.

Знати: основні види сховищ мобільної платформи та їх призначення; API мобільної платформи, що дозволяє використовувати локальні та захищені сховища; метод і інтернаціоналізацію мобільного додатка.

Вміти: програмувати функціональні вимоги до мобільного пристрою, що потребують зберігання персональних даних у локальні та захищені сховища та виконувати інтернаціоналізацію мобільного додатка; ефективно формувати комунікаційну стратегію під час роботи в команді.

Нести відповідальність за якість реалізації функціональних вимог до мобільного додатка та строків її реалізації.

Основні питання:

1. Огляд постачальників контенту.
2. Робота з постачальником контенту.
3. Завантажувачі (`Loaders`).
4. Постачальник контактів.
5. Вбудовані платформи доступу до сховищ.

Ключові слова: атрибут, постачальники контенту, контракт, SQLight, API-інтерфейс, адаптери синхронізації, методанні.

Постачальники контенту (Content providers) управляють доступом до структурованого набору даних. Вони інкапсулюють дані й надають механізми забезпечення їхньої безпеки. Постачальники контенту – це стандартний інтерфейс для об'єднання даних в одному процесі з кодом, який виконується в іншому процесі [2; 3; 16].

Коли потрібен доступ до даних у постачальнику контенту, треба використовувати об'єкт `ContentResolver` в інтерфейсі `Context` програмного додатка, щоб підключитися до постачальника як клієнт. Об'єкт `Content Resolver` взаємодіє з об'єктом постачальника, який є екземпляром класу, який реалізує об'єкт `ContentProvider`. Об'єкт постачальника отримує від клієнтів запити даних, виконує запрошувані дії та повертає результати.

Не потрібно розробляти власний постачальник, якщо не планують надавати доступ до своїх даних іншим програмам. Однак знадобиться власний постачальник для надання налаштованих пошукових підказок у власному додатку. Також знадобиться власний постачальник, якщо потрібно копіювати та вставляти складні дані або файли зі свого програмного додатка в інші.

До складу системи Android входять постачальники контенту, які управляють такими даними, як аудіо-, відеозображення та особиста контактна інформація. Деякі з постачальників указані в довідковій документації для пакета `android.provider`. Працювати із цими постачальниками може будь-який додаток Android (проте з деякими обмеженнями) [2; 3; 16].

4.1. Основні відомості про постачальника контенту

Постачальник контенту управляє доступом до центрального сховища даних. Постачальник є компонентом програми Android, який часто має власний інтерфейс користувача для роботи з даними. Однак постачальники контенту призначено, у першу чергу, для використання іншими програмними додатками, які отримують доступ до постачальника за допомогою клієнтського об'єкта постачальника. Разом постачальники та клієнти постачальників забезпечують погоджений, стандартний інтерфейс доступу до даних, що також обробляє взаємодію між процесами та забезпечує захищений доступ до даних.

Постачальник контенту надає дані зовнішнім програмним додаткам у вигляді однієї або декількох таблиць, аналогічних таблицям у реляційній базі даних. Рядок становить екземпляр деякого типу даних, що збираються постачальником, а кожен стовпець у цьому рядку – це окремий елемент даних, зібраних для екземпляра.

Прикладом вбудованого постачальника у платформі Android може бути користувальницький словник, у якому зберігаються дані про написання нестандартних слів, доданих користувачем. У табл. 4.1 показано, який вигляд можуть мати дані в цій таблиці постачальника.

Таблиця 4.1

Приклад таблиці словника

word	app id	frequency	locale	_ID
1	2	3	4	5
mapreduce	user1	100	en_US	1

Закінчення табл. 4.1

1	2	3	4	5
precompiler	user14	200	fr_FR	2
applet	user2	225	fr_CA	3
const	user1	255	pt_BR	4
int	user5	100	en_UK	5

У кожному рядку табл. 4.1 наведено екземпляр слова, відсутнього у стандартному словнику. У кожному її стовпці містяться деякі дані для слова, наприклад, дані про мову, на якій це слово було вперше використано. Заголовки стовпців – це назва стовпців, які зберігаються у постачальнику. Щоб дізнатися мову рядка, необхідно звернутися до стовпця locale. У цьому постачальнику стовпець _ID є *основним ключем*, який постачальник автоматично зберігає.

Примітка. У постачальника необов'язково має бути основний ключ, а також йому необов'язково використовувати `_ID` як назву стовпця основного ключа, якщо такий є. Однак, якщо необхідно прив'язати дані з постачальника до класу `ListView`, один із стовпців має називатися `_ID`.

4.1.1. Доступ до постачальника

Для доступу програми до даних із постачальника контенту використовують клієнтський об'єкт `ContentResolver`. У цьому об'єкті є методи, які викликають ідентичні методи в об'єкті постачальника, що є екземпляром одного з конкретних підкласів класу `ContentProvider`. У цих методах `ContentResolver` подано основні функції CRUD (абревіатура create, retrieve, update, delete [створення, отримання, оновлення та видалення]) постійного сховища.

Об'єкт `ContentResolver` у процесі клієнтської програми й об'єкт `ContentProvider` у програмі, яка володіє постачальником, автоматично обробляють взаємодію між процесами. Об'єкт `ContentProvider` також є рівнем абстракції між репозиторієм даних і зовнішнім уявленням даних у вигляді таблиць.

Примітка. Для доступу до постачальника програма зазвичай має запросити певні дозволи у своєму файлі маніфесту.

Наприклад, щоб отримати з постачальника користувальницького словника список слів і мов, на яких їх уявлення наведено, слід викликати метод `ContentResolver.query()`. У свою чергу, метод `query()` викликає метод `ContentProvider.query()`, визначений постачальником словника.

У прикладі коду рис. 4.1 показано виклик методу `ContentResolver.query()`.

```
// Запит до словника користувача та отримання результату
mCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,   // URI слів таблиці
    mProjection,                        // Колонки для кожного рядка
    mSelectionClause,                   // Критерій вибірки
    mSelectionArgs,                     // Критерій вибірки
    mSortOrder);                        // Порядок для виведення рядків
```

Рис. 4.1. **Викликання методу `ContentResolver.query()`**

У табл. 4.2 зазначено відповідність аргументів для методу запиту `query(Uri, projection, selection, selectionArgs, sortOrder)` SQL-команди SELECT.

Таблиця 4.2

Порівняння методу `query()` і SQL-запиту

Аргумент методу `query()`	Параметр / ключове слово `SELECT`	Примітки
`Uri`	`FROM` *table_name*	`Uri` відповідає таблиці *table_name* у постачальнику
`projection`	*col*,*col*,*col*,...	`projection` – це масив стовпців, які необхідно включити в кожний отриманий рядок
`selection`	`WHERE` *col* `=` value	`selection` задає критерії для вибору рядків
`selectionArgs`	(Точний еквівалент відсутній. Заповнювачі **?** замінюються аргументами вибору)	–
`sortOrder`	`ORDER BY` *col*,*col*,...	`sortOrder` задає порядок відображення рядків в об'єкті `Cursor`

4.1.2. URI контенту

URI контенту – це URI, який визначає дані в постачальнику. URI контенту можуть включати символічну назву всього постачальника (його **центр**) і назву, яка вказує на таблицю (**шлях**). Під час виклику клієнтського

методу для доступу до таблиці в постачальнику URI контенту цієї таблиці є одним з аргументів цього методу.

Константа `CONTENT_URI` в попередніх рядках коду містить URI контенту таблиці words у користувальницькому словнику. Об'єкт `ContentResolver` аналізує центр URI і використовує його для "вирішення" постачальника шляхом порівняння центру із системною таблицею відомих постачальників. `ContentResolver` може відправити аргументи запиту до відповідного постачальника.

`ContentProvider` використовує частину URI контенту, в якій указано шлях вибору таблиці для доступу. У постачальника зазвичай є шлях для кожної наданої ним таблиці.

У попередніх рядках коду повний URI для таблиці words має такий вигляд (рис. 4.2):

```
content://user_dictionary/words
```

Рис. 4.2. **Повний URI для таблиці words**

Рядок `user_dictionary` – це центр постачальника, а рядок `words` – це шлях до таблиці. Рядок `content://` (**схема**) наявний завжди; він визначає, що це URI контенту.

Багато постачальників надають доступ до одного рядка в таблиці шляхом додавання ідентифікатора в кінець URI. Наприклад, щоб витягти зі словника рядок, у стовпці `_ID` якого задано 4, можна скористатися такими URI контенту (рис. 4.3):

```
Uri singleUri =
ContentUris.withAppendedId(UserDictionary.Words.CONTENT_URI,4);
```

Рис. 4.3. **Витягування зі словника рядок**

Ідентифікатори часто використовують у випадку, коли витягнуто набір рядків і потрібно оновити або видалити один із них.

Примітка. У класах `Uri` і `Uri.Builder` є методи для зручного створення правильно оформлених об'єктів URI із рядків. `ContentUris` містить методи для зручного додавання ідентифікаторів до URI. У прикладі коду раніше для додавання ідентифікатора до URI контенту `UserDictionary` використано метод `withAppendedId()`.

4.1.3. Отримання даних від постачальника

У цьому підрозділі розглядається порядок отримання даних від постачальника на прикладі постачальника *користувальницького словника*.

Для повної ясності у прикладах коду, наведених у цьому підрозділі, методи `ContentResolver.query()` викликаються в потоці призначеного для інтерфейсу користувача. У реальному коді запити слід виконувати асинхронно в окремому потоці. Одним із способів реалізувати це є використання класу `CursorLoader`.

Щоб отримати дані з постачальника, слід виконати такі основні дії:

1. Запитати в постачальника дозвіл на читання.

2. Визначити код, який відповідає за відправлення запиту постачальнику.

Запит дозволу на читання

Щоб ваша програма могла отримувати дані від постачальника, програмі слід отримати від постачальника дозвіл на читання. Цей дозвіл неможливо отримати під час виконання; замість цього необхідно вказати у маніфесті програми, що потрібен такий дозвіл. Для цього слід скористатися елементом `<uses-permission>` і вказати точну назву дозволу, зазначену постачальником. Указавши цей елемент у маніфесті, тим самим запитують необхідний дозвіл для програми. Коли користувачі встановлюють програму, вони отримують дозвіл на цей запит.

Щоб дізнатися точну назву дозволу на читання у використовуваному постачальнику, а також назви інших використовуваних у ньому дозволів на читання, треба звернутися до документації постачальника.

Постачальник *користувальницького словника* задає дозвіл `android.permission.READ_USER_DICTIONARY` у своєму файлі маніфесту, тому програмі, якій потрібно виконати читання даних із постачальника, необхідно запросити саме цей дозвіл.

Створення запиту

Наступним етапом отримання даних від постачальника є створення запиту. У такому фрагменті коду задано деякі змінні для доступу до постачальника словника (рис. 4.4):

```
// projection – це масив стовпців, які необхідно включити в кожний отри-
маний рядок
String[] mProjection =
{
    UserDictionary.Words._ID,    // Константа для сповпця _ID
    UserDictionary.Words.WORD,   // Константа для сповпця word
    UserDictionary.Words.LOCALE  // Константа для сповпця locale
};

// Визначає рядок, що містить умову вибору
String mSelectionClause = null;

// Визначає масив, що містить аргументи для вибірки
String[] mSelectionArgs = {""};
```

Рис. 4.4. **Задавання деяких змінних для доступу до постачальника словника**

У наступному фрагменті коду продемонстровано порядок використання методу `ContentResolver.query()` (прикладом є постачальник словника): клієнтський запит постачальника аналогічний SQL-запиту. У ньому міститься набір стовпців, які повертаються, набір критеріїв вибірки та порядок сортування.

Набір стовпців, які має повернути запит, називається **проекцією** (змінна `mProjection`).

Вираз, який задає рядки для отримання, складається із пропозиції вибору та аргументів вибору. Пропозиція вибору становить поєднання логічних виразів, назв стовпців і значень (змінна `mSelectionClause`). Якщо замість значення вказати підставлюваний параметр `?`, метод запиту витягує значення з масиву аргументів вибору (змінна `mSelectionArgs`).

У наступному фрагменті коду, якщо користувач не вказав слово, то для пропозиції вибору задають значення `null`, а запит повертає всі слова, наявні у постачальника. Якщо користувач указав слово, то для пропозиції вибору задають значення `UserDictionary.Words.WORD + "=?"`, а для першого елемента в масиві аргументів вибору задано введене користувачем слово (рис. 4.5):

```
/* Завдається одноелементний масив типу String, що містить аргументи
вибірки.  */
String[] mSelectionArgs = {""};

// Отримуємо слово з UI
mSearchString = mSearchWord.getText().toString();
// Не забудьте вставити тут код для перевірки даних на дійсність.

// Дії, якщо слово є порожнім рядком
if (TextUtils.isEmpty(mSearchString)) {
  // Установлення значення null для змінної вибірки, таким чином резуль-
татом будуть усі можливі слова
  mSelectionClause = null;
  mSelectionArgs[0] = "";

} else {
  // Установлює умову вибору, що відповідає слову, що ввів користувач.
  mSelectionClause = UserDictionary.Words.WORD + " = ?";

  // Переміщення рядка, що ввів користувач до аргументів вибірки.
  mSelectionArgs[0] = mSearchString;
}

// Запит до таблиці та повернення об'єкту типу Cursor
mCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,  // URI слів таблиці
    mProjection,       // Колонки для кожного рядка
    mSelectionClause   // або null, або слово, що ввів користувач
    mSelectionArgs,    // або пустий рядок, або рядок, що ввів користувач
    mSortOrder);       // Порядок для виведення рядків

// Деякі провайдери повертають нульове значення, якщо відбувається поми-
лка, інші кидають виняткову ситуацію
if (null == mCursor) {
  /* Уставте тут код для оброблення виняткової ситуації.
Не використовуйте курсор! Ви можна викликати android.util.Log.e,
щоб записати інформацію про цю виняткову ситуацію.  */

// Якщо курсор пустий, то провайдер не знаходить збігів
} else if (mCursor.getCount() < 1) {
```

Рис. 4.5. **Реалізована проекція**

```
  /* Уставте тут код, щоб повідомити користувача, що пошук не виконався
успішно. Це не обов'язково. */

} else {
  // Уставте код тут, щоб обробити результати
}
```

Закінчення рис. 4.5

Цей запит аналогічний SQL-інструкції (рис. 4.6):

```
SELECT _ID, word, locale FROM words WHERE word = <userinput> ORDER BY
word ASC;
```

Рис. 4.6. **Запит аналогічний SQL-інструкції**

У цій інструкції SQL, замість констант класу-контракту, використовують фактичні назви стовпців.

Захист від уведення шкідливого коду. Якщо дані, якими управляє постачальник контенту, знаходяться в базі даних SQL, то включення в необроблені інструкції SQL зовнішніх ненадійних даних може призвести до атаки шляхом уставлення коду SQL.

Слід розглянути таку умову на вибірку (рис. 4.7):

```
// Складаємо умову на вибірку шляхом складання змінної, що ввів користу-
вач та назви стовпця
String mSelectionClause =  "var = " + mUserInput;
```

Рис. 4.7. **Умова на вибірку**

Якщо використовувати цю умову, дозволено користувачеві пов'язати інструкцію SQL зі шкідливим кодом SQL. Наприклад, користувач може ввести nothing; DROP TABLE *; для `mUserInput`, що приведе до створення такої умови вибору: `var = nothing; DROP TABLE * ;`. Оскільки умова вибору виконується в потоці як інструкція SQL, це може призвести до того, що постачальник видалить усі таблиці у відповідній базі даних SQLite (якщо тільки постачальник не налаштовано на відстеження спроб установити шкідливий код SQL).

Щоб уникнути цього, слід скористатися умовою вибору, у якій `?` є підставлюваним параметром, а також окремим масивом аргументів вибору. Після цього введення даних користувачем буде пов'язано безпосередньо із запитом і не буде інтерпретуватися як частина інструкції SQL. Оскільки в цьому випадку запит, що ввів користувач, не буде розглядатися як код SQL, то в нього не вдасться запровадити шкідливий код SQL. Замість об'єднання, яке варто включити в *користувальницький ввід даних*, слід використовувати таку умову вибору (рис. 4.8):

```
// Створює пункт вибору зі змінним параметром
String mSelectionClause =  "var = ?";
```

Рис. 4.8. **Умова вибору, замість об'єднання**

Налаштувати масив аргументів вибору таким чином (рис. 4.9):

```
// Defines an array to contain the selection arguments
String[] selectionArgs = {""};
```

Рис. 4.9. **Масив аргументів вибору**

Указати значення для масиву аргументів вибору (рис. 4.10):

```
// Sets the selection argument to the user's input
selectionArgs[0] = mUserInput;
```

Рис. 4.10. **Значення для масиву аргументів вибору**

Умова вибору, у якій `?` використано як підставлюваний параметр, і масив аргументів вибору є кращим способом указання вибору, навіть якщо постачальник не використовує базу даних SQL.

Відображення результатів запиту

Клієнтський метод `ContentResolver.query()` завжди повертає об'єкт Cursor, що містить стовпці, зазначені у проекції запиту для рядків, які відповідають критеріям вибірки в запиті. Об'єкт `Cursor` надає прямий доступ до читання рядків і стовпців, що містяться в ньому. За допомогою методів `Cursor` можна виконати ітерацію по рядках у результатах, визначити тип даних для кожного стовпця, отримати дані з рядка, а також перевірити

інші властивості результатів. Деякі реалізації об'єкта `Cursor` автоматично оновлюють об'єкт під час зміни даних у постачальнику або запускають виконання методів в об'єкті-спостерігачі під час зміни об'єкта Cursor або виконують і те, і інше.

Примітка. Постачальник може обмежити доступ до стовпців на основі характеру об'єкта, що виконує запит. Наприклад, постачальник контактів обмежує доступ адаптерів синхронізації до деяких стовпців, тому він не повертає їх у дію або службу.

Якщо рядки, які відповідають критеріям вибірки, відсутні, постачальник повертає об'єкт `Cursor` у якому для методу `Cursor.getCount()` указано значення "`0`" (порожній об'єкт `cursor`).

Під час виникнення внутрішньої помилки результати запиту залежать від певного постачальника. Постачальник може повернути `null` або видати `Exception`.

Оскільки `Cursor` становить "список" рядків, то найкращим способом відобразити вміст об'єкта `Cursor` буде пов'язати його з `ListView` за допомогою `SimpleCursorAdapter`.

Наступний фрагмент коду є продовженням попереднього фрагмента. Він створює об'єкт `SimpleCursorAdapter`, що містить об'єкт `Cursor`, який був отриманий у запиті, а потім визначає цей об'єкт як адаптер для `ListView` (рис. 4.11):

```
// Визначає список стовпців для отримання з курсора та завантаження
// в рядок виведення
String[] mWordListColumns =
{
    UserDictionary.Words.WORD,   // містить назву стовпця word
    UserDictionary.Words.LOCALE  // містить назву стовпця locale
};

// Визначає список View IDs, які будуть отримувати стовпці курсора
// для кожного рядка
int[] mWordListItems = { R.id.dictWord, R.id.locale};

// Створює новий SimpleCursorAdapter
mCursorAdapter = new SimpleCursorAdapter(
    getApplicationContext(),  // Об'єкт Context програми
    R.layout.wordlistrow,     // Шаблон у XML для одного рядка в ListView
```

Рис. 4.11. **Продовження попереднього фрагмента**

```
    mCursor,                  // Результат запиту
    mWordListColumns,         // Масив назв колонок у курсорі
    mWordListItems,           // Масив view IDs
   0);                   // Прапорці (зазвичай немає в них необхідності)

// Встановлюється адаптер для the ListView
mWordList.setAdapter(mCursorAdapter);
```

Закінчення рис. 4.11

Примітка. Щоб повернути `ListView` з об'єктом `Cursor`, цей об'єкт `cursor` має містити стовпець з назвою `_ID`. Тому показаний раніше запит отримує стовпець `_ID` для таблиці words, навіть якщо `ListView` не відображає її. Це обмеження також пояснює, чому в кожній таблиці постачальника є стовпець `_ID`.

Отримання даних із результатів запиту. Замість того, щоб просто переглянути результати запиту, можна використовувати їх для виконання інших завдань. Наприклад, можна отримати написання слів зі словника користувача, а потім виконати їх пошук в інших постачальниках. Для цього слід виконати ітерацію по рядках в об'єкті `Cursor` (рис. 4.12):

```
// Визначаємо індекс стовпця, що має назву word
int index = mCursor.getColumnIndex(UserDictionary.Words.WORD);
/*
* Виконується, якщо курсор є валідним. Провайдер словника користувача
* повертає нульове значення, якщо виникла внутрішня помилка.
* Інші провайдери можуть повернути помилку, замість повернення нуля.
*/

if (mCursor != null) {
  /*
  * Перехід до наступного рядка в курсорі. Перед першим переміщенням у
  * курсорі "Індекс рядка" -1, і якщо ви намагаєтеся отримати дані в цій
  * позиції, то ви отримуєте виняткову ситуацію (помилку)
  */
  while (mCursor.moveToNext()) {
    // Отримуємо значення стовпця.
    newWord = mCursor.getString(index);
    // Уставте тут код для оброблення отриманого слова
```

Рис. 4.12. **Ітерація по рядках в об'єкті `Cursor`**

```
    ...
    // кінець циклу
  }
} else {

    // Уставте тут код для звіту про помилку, якщо курсор є null або
    // провайдер викинув виняткову ситуацію.
}
```

Закінчення рис. 4.12

Реалізації об'єкта `Cursor` містять кілька методів get для отримання об'єкта різних типів даних. Наприклад, у наступному фрагменті коду використовують метод `getString()`. У них також є метод `getType()`, що повертає значення, яке вказує на тип даних стовпця.

4.1.4. Дозволи постачальника контенту

Додаток постачальника може задавати дозволи, які потрібні іншим додаткам для доступу до даних у постачальника. Такі дозволи гарантують, що користувач знає, до яких даних додатка буде намагатися отримати доступ. На основі вимог постачальника інші програми запитують дозволи, які потрібні їм для доступу до постачальника. Кінцеві користувачі бачать запитані дозволи під час установлення програми.

Якщо додаток постачальника не задає ніяких дозволів, інші додатки не отримують доступу до даних постачальника. Однак до компонентів додатка постачальника завжди надано повний доступ на читання та запис, незалежно від заданих дозволів.

Як уже було зазначено раніше, для отримання даних із постачальника користувальницького словника потрібен дозвіл `android.permission.READ_USER_DICTIONARY`. У постачальника передбачено окремий дозвіл `android.permission.WRITE_USER_DICTIONARY` для вставлення, оновлення або видалення даних. Щоб отримати необхідні дозволи для доступу до інтернету, програма запитує їх за допомогою елемента `<uses-permission>` у файлі маніфесту. Під час установлення менеджером пакетів Android програми користувачеві необхідно затвердити всі дозволи, необхідні додатку. У разі затвердження всіх дозволів менеджер пакетів продовжує установлення; якщо ж користувач відхиляє їх, менеджер пакетів скасовує установлення.

Для запиту доступу до читання даних у постачальника користувальницького словника використовують елемент `<uses-permission>` (рис. 4.13):

```
<uses-permission android:name="android.permission.READ_USER_DICTIONARY">
```

Рис. 4.13. **Елемент** `<uses-permission>`

4.1.5. Уставлення, оновлення та видалення даних

Подібно до того, як отримують дані від постачальника, також можна використовувати можливості взаємодії між клієнтом постачальника та об'єктом `ContentProvider` постачальника для зміни даних. Можна викликати метод об'єкта `ContentResolver`, указавши аргументи, які були передані у відповідний метод об'єкта `ContentProvider`. Постачальник і клієнт постачальника автоматично обробляють взаємодію між процесами та забезпечують безпеку.

Уставлення даних

Для вставлення даних у постачальник слід викликати метод `ContentResolver.insert()`. Цей метод уставляє новий рядок у постачальник і повертає URI контенту для цього рядка. У такому фрагменті коду продемонстровано порядок уставлення нового слова в постачальник користувальницького словника (рис. 4.14):

```
// Визначаємо новий об'єкт Uri, який отримує результат уставлення
Uri mNewUri;
// Визначаємо об'єкт, який містить нові значення для вставлення
ContentValues mNewValues = new ContentValues();

// Установлюємо значення кожного стовпця та вставляємо слово.
mNewValues.put(UserDictionary.Words.APP_ID, "example.user");
mNewValues.put(UserDictionary.Words.LOCALE, "en_US");
mNewValues.put(UserDictionary.Words.WORD, "insert");
mNewValues.put(UserDictionary.Words.FREQUENCY, "100");

mNewUri = getContentResolver().insert(
   UserDictionary.Word.CONTENT_URI, // URI контенту словника користувача
    mNewValues                      // значення для вставлення
);
```

Рис. 4.14. **Уставлення нового слова в постачальник користувальницького словника**

Дані для нового рядка надходять в один об'єкт `ContentValues`, який аналогічний об'єкту `cursor` з одним рядком. Стовпці в цьому об'єкті необов'язково мають містити дані такого ж типу, і якщо не потрібно вказувати значення, можна задати для стовпця значення `null` за допомогою методу `ContentValues.putNull()`.

Код у наведеному фрагменті не додає стовпець `_ID`, оскільки цей стовпець зберігається автоматично. Постачальник надає унікальне значення `_ID` кожному доданому рядку. Зазвичай постачальники використовують це значення як основного ключа таблиці.

URI контенту, повернений в елементі `newUri`, слугує для ідентифікації нового доданого рядка в такому форматі (рис. 4.15):

```
content://user_dictionary/words/<id_value>
```

Рис. 4.15. **Ідентифікації нового доданого рядка**

`<id_value>` – це вміст стовпця `_ID` для нового рядка. Більшість постачальників автоматично визначають цю форму URI контенту, а потім виконують необхідну операцію з відповідним рядком.

Щоб отримати значення `_ID` з повернутого об'єкта `Uri`, слід викликати метод `ContentUris.parseId()`.

Оновлення даних

Щоб оновити рядок, слід використовувати об'єкт `ContentValues` з оновленими значеннями (точно так само, як це роблять під час уставлення) та критеріями вибірки (так само, як і з запитом). Використовуваний клієнтський метод названо `ContentResolver.update()`. Не потрібно додавати значення в об'єкт `ContentValues` для оновлюваних стовпців. Щоб очистити вміст стовпця, слід установити значення `null`.

Наведений фрагмент коду слугує для зміни мови у всіх рядках, де як мову зазначено `en`, на значення `null`. Обчислене значення – це кількість рядків, які було оновлено (рис. 4.16).

Також слід перевірити введені користувачем дані під час виклику методу `ContentResolver.update()`.

```
// Визначаємо об'єкт, який містить оновлені значення
ContentValues mUpdateValues = new ContentValues();

// Визначаємо критерії вибору для рядків, які ви хочете оновити
String mSelectionClause = UserDictionary.Words.LOCALE +  "LIKE ?";
String[] mSelectionArgs = {"en_%"};

// Визначаємо змінну, щоб утримувати кількість оновлених рядків
int mRowsUpdated = 0;

/*
 * Установлюємо оновлене значення та оновлюємо обрані слова.
 */
mUpdateValues.putNull(UserDictionary.Words.LOCALE);

mRowsUpdated = getContentResolver().update(
    UserDictionary.Words.CONTENT_URI,   // URI контенту словника корис-
тувача
    mUpdateValues                       // колонка для оновлення даних
    mSelectionClause                    // стовпець, за яким вибирають
    mSelectionArgs                      // значення для порівняння
);
```

Рис. 4.16. **Зміна мови у всіх рядках, де як мову зазначено `en`, на значення `null`**

Видалення даних

Видалення даних аналогічно отриманню даних рядка: необхідно вказати критерії вибірки для рядків, які потрібно видалити, після чого клієнтський метод поверне кількість видалених рядків. Далі наведено фрагмент коду для видалення рядків з ідентифікатором *appid user*. Метод повертає кількість видалених рядків (рис. 4.17).

Також слід перевірити дані введені користувачем під час виклику методу `ContentResolver.delete()`.

```
// Визначаємо критерії вибору для рядків, які ви хочете видалити
   String mSelectionClause = UserDictionary.Words.APP_ID + " LIKE ?";
   String[] mSelectionArgs = {"user"};
```

Рис. 4.17. **Видалення рядків з ідентифікатором appid user**

```
// Визначаємо змінну, щоб отримувати кількість видалених рядків
     int mRowsDeleted = 0;
...
// Видаляє слова, які відповідають критеріям відбору
    mRowsDeleted = getContentResolver().delete(
    UserDictionary.Words.CONTENT_URI,   // URI контенту словника корис-
тувача
    mSelectionClause                    // стовпець, за яким вибирають
    mSelectionArgs                      // значення для порівняння
);.
```

Закінчення рис. 4.17.

4.1.6. Типи даних постачальників

Постачальники контенту можуть надавати різні типи даних. Постачальник користувальницького словника надає тільки текст, але також може надавати такі формати:

- ціле число;
- довге ціле число (long);
- число з плаваючою комою;
- довге число з рухомою(плаваючою) комою (double).

Іншим типом даних, що пропонуються постачальником, є великий двійковий об'єкт (BLOB), реалізований як 64-розрядний масив. Щоб переглянути доступні типи даних, слід звернутися до методів get класу `Cursor`.

Тип даних для кожного стовпця в постачальнику зазвичай вказано в документації до постачальника. Типи даних для постачальника користувальницького словника вказано в довідковій документації для класу-контракту `UserDictionary.Words`. Також визначити тип даних можна шляхом виклику методу `Cursor.getType()`.

Постачальники також зберігають інформацію про тип даних MIME для кожного обумовленого ними URI контенту. Цю інформацію можна використовувати для визначення того, чи може додаток обробляти пропоновані постачальником дані, а також для вибору типу оброблення на основі типу MIME. Інформація про тип MIME зазвичай потрібна під час роботи з постачальником, який містить складні структури даних або файли. Наприклад, у таблиці `ContactsContract.Data` в постачальника контактів використовують типи MIME для позначення типу даних контакту, які

зберігаються в кожному рядку. Щоб отримати тип MIME, відповідний URI контенту, слід викликати метод `ContentResolver.getType()`.

Синтаксис стандартних і налаштованих типів MIME описано в довідці за типами MIME.

4.1.7. Альтернативні форми доступу до постачальника

Під час розроблення програми варто враховувати три альтернативних форми доступу до постачальника:

• Пакетний доступ: можна створити пакет викликів доступу з використанням методів у класі `ContentProviderOperation`, а потім застосувати їх за допомогою методу `ContentResolver.applyBatch()`.

• Асинхронні запити: запити слід виконувати в окремому потоці. Одним із способів реалізувати це є використання об'єкта `CursorLoader`.

• Доступ до даних за допомогою намірів: незважаючи на те що намір неможливо відправити безпосередньо в постачальник, можна відправити запит на додаток постачальника, у якому зазвичай є більше можливостей для зміни даних постачальника.

Пакетний доступ і зміна з допомогою намірів описано в наступних підрозділах.

Пакетний доступ

Пакетний доступ до постачальника корисно використовувати у випадках, коли необхідно вставити велику кількість рядків або для вставлення рядків в кілька таблиць у межах одного виклику методу, а також у загальних випадках для виконання ряду операцій на кордонах процесів у вигляді транзакції (атомарної операції).

Для доступу до постачальника в "пакетному режимі" необхідно створити масив об'єктів `ContentProviderOperation`, а потім відправити їх у постачальник контенту за допомогою методу `ContentResolver.applyBatch()`. У цей метод необхідно передати *центр* постачальника контенту, а не певний URI контенту. Це дозволить кожному об'єкту `ContentProviderOperation` у масиві взаємодіяти з різними таблицями. Метод `ContentResolver.applyBatch()` повертає масив результатів.

В описі класу-контракту `ContactsContract.RawContacts` також наведено фрагмент коду, у якому продемонстровано вставку в пакетному режимі. У вихідному файлі `ContactAdder.java` прикладу програми *Диспетчер контактів* є приклад пакетного доступу.

Доступ до даних за допомогою намірів

Наміри дозволяють в обхід отримувати доступ до свого контенту. Можна дозволити користувачам доступ до даних у постачальника навіть у тому випадку, якщо в додатках відсутні дозволи на доступ, або шляхом отримання результівного наміру від програми, у якої є необхідні дозволи, або шляхом активації програми, у якої є дозвіл і яка дозволяє користувачеві працювати з ним.

Отримання доступу з тимчасовими дозволами. Можна отримати доступ до даних у постачальника контенту навіть тоді, коли немає необхідних дозволів на доступ, шляхом надсилання наміру в додаток, у якого є такі дозволи, та отримання результівного наміру, який містить дозволи URI. Ці дозволи для певного URI контенту діють до тих пір, поки не буде завершено операцію, яка отримала їх. Додаток, у якого є безстрокові дозволи, надає тимчасові дозволи шляхом установлення відповідного прапорця в результівному намірі:

- Дозвіл на читання: `FLAG_GRANT_READ_URI_PERMISSION`.
- Дозвіл на запис: `FLAG_GRANT_WRITE_URI_PERMISSION`.

Примітка. Ці прапорці не надають доступ до читання або запису постачальнику, центр якого вказано в URI контенту. Доступ надається тільки самому URI.

Постачальник визначає дозволи URI для URI контенту у своєму маніфесті з допомогою атрибута `android:grantUriPermission` елемента `<provider>`, а також за допомогою дочірнього елемента `<grant-uri-permission>` елемента `<provider>`.

Наприклад, можна отримати дані про контакт із постачальника контактів, навіть якщо немає дозволу `READ_CONTACTS`. Можливо, це потрібно реалізувати в додатку, який відправляє електронні привітання контакту в день його народження. Замість запиту `READ_CONTACTS`, коли можна отримати доступ до всіх контактів користувача та інформацію про них, слід надати користувачеві можливість указати, які контакти використовуються додатком. Для цього слід скористатися наведеними далі процесами.

1. Програма відправляє намір, що містить дію `ACTION_PICK` і тип MIME `CONTENT_ITEM_TYPE` контактів, використовуючи для цього метод `startActivityForResult()`.

2. Оскільки цей намір відповідає умовам відбору намірів для операції вибору програми *Контакти*, ця операція переходить на передній план.

3. В операції вибору користувач вибирає контакт для оновлення. Коли це відбувається, операція вибору викликає метод `setRe-sult (resultcode, intent)` для створення наміру, який буде передано назад у програму. Намір містить URI вмісту вибраного користувачем контакту, а також прапорці `FLAG_GRANT_READ_URI_PERMISSION` додаткових даних. Ці прапорці надають додатку дозвіл URI на читання даних контакту, на який указує URI контенту. Потім операція вибору викликає метод `finish()`, щоб повернути управління додатком.

4. Операція повертається на передній план, а система викликає метод `onActivityResult()` операції. Цей метод отримує результівний намір, створений операцією вибору у програмі *Контакти*.

5. За допомогою URI контенту з результівного наміру можна виконати читання даних контакту з постачальника контактів, навіть якщо не запитували в постачальника постійний доступ до читання у своєму маніфесті. Можна отримати інформацію про день народження контакту або відомості про його адресу електронної пошти, а потім відправити контакту електронне привітання.

Використання іншого додатка. Простий спосіб дозволити користувачеві змінювати дані, на доступ до яких немає дозволу, – це активувати додаток, у якого є такі дозволи, а потім надати користувачеві можливість виконувати необхідні дії в цьому додатку.

Наприклад, додаток *Календар* приймає намір `ACTION_INSERT`, за допомогою якого можна активувати інтерфейс користувача програми для вставлення. Можна передати в цьому намірі додаткові дані, які додаток використовує для заповнення полів в інтерфейсі. Оскільки синтаксис повторюваних подій досить складний, то події слід уставляти в постачальник календаря шляхом активації програми *Календар* за допомогою дії `ACTION_INSERT` і подальшого надання користувачеві можливості самому вставити подію в цей додаток.

Відображення даних за допомогою допоміжного додатка. Якщо програмі не надано дозволу, як і раніше можна скористатися наміром для відображення даних в іншому додатку. Наприклад, додаток *Календар* приймає намір `ACTION_VIEW`, який дозволяє відобразити певну дату або подію. Завдяки цьому інформацію календаря можна відображати без необхідності створювати власний інтерфейс користувача.

Додаток, у який надсилають намір, не обов'язково має бути пов'язано з постачальником. Наприклад, у постачальника контактів можна

створити форму контакту, а потім відправити намір `ACTION_VIEW`, що містить URI контенту для зображення контакту, засіб перегляду зображень.

4.1.8. Класи-контракти

Клас-контракт визначає константи, які забезпечують для додатків можливість працювати з URI контенту, назвами стовпців, операціями намірів та іншими функціями постачальника вмісту. Класи-контракти не включено в постачальника; розробнику постачальника слід визначити їх і зробити доступними для інших розробників. Багато хто з постачальників, включених у платформу Android, містять відповідні класи-контракти в пакеті `android.provider`.

Наприклад, у постачальника користувальницького календаря є клас-контракт `UserDictionary`, що містить константи URI контенту та назв стовпців. URI вмісту таблиці words визначено в константі `UserDictionary.Words.CONTENT_URI`. У класі `UserDictionary.Words` також є константи назв стовпців, які використовують у фрагментах коду прикладу програми. Наприклад, проекцію запиту можна визначити таким чином (рис. 4.18):

```
String[] mProjection =
{
    UserDictionary.Words._ID,
    UserDictionary.Words.WORD,
    UserDictionary.Words.LOCALE
};
```

Рис. 4.18. **Проекція запиту**

Іншим класом-контрактом є клас `ContactsContract` для постачальника контактів. Один із його підкласів, `ContactsContract.Intents.Insert`, становить клас-контракт, який містить константи для намірів і їхніх даних.

4.1.9. Довідка за типами MIME

Постачальники контенту можуть повертати як стандартні типи мультимедіа MIME, так і рядки з налаштованим типом MIME, або обидва ці типи.

Типи MIME мають такий формат (рис. 4.19):

```
type/subtype
```

Рис. 4.19. **Формат типів MIME**

Наприклад, добре відомий тип MIME `text/html` має тип text підтип html. Якщо постачальник повертає цей тип URI, це означає, що рядок запиту, у якому використовують цей URI, поверне текст із тегами HTML.

Рядки з налаштованим типом MIME, які також називають типами MIME постачальника, мають більш складні значення типів і підтипів. Значення типу завжди таке (рис. 4.20):

```
vnd.android.cursor.dir
```

а

```
vnd.android.cursor.item
```

б

Рис. 4.20. **Значення типу для декількох рядків (а) і для одного рядка (б)**

Підтип залежить від постачальника. Вбудовані постачальники Android зазвичай містять простий підтип. Наприклад, коли програма "Контакти" створює рядок для номера телефону, вона задає такий тип MIME в цьому рядку (рис. 4.21):

```
vnd.android.cursor.item/phone_v2
```

Рис. 4.21. **Тип MIME, який задає програма "Контакти"**

Зверніть увагу, що значення підтипу просто `phone_v2`.

Розробники постачальників можуть створювати свої власні шаблони підтипів на основі центру та назв таблиць постачальника. Наприклад, слід розглянути постачальник, який містить розклад руху поїздів. Центром постачальника є `com.example.trains`, у якому містяться таблиці Line1, Line2 і Line3.

А. У відповідь на такий URI вмісту (рис. 4.22) для таблиці Line1 постачальник повертає такий тип MIME (рис. 4.23).

```
content://com.example.trains/Line1
```

Рис. 4.22. **URI вмісту**

```
vnd.android.cursor.dir/vnd.example.line1
```

Рис. 4.23. **Тип MIME для таблиці Line1**

В. У відповідь на такий URI вмісту (рис. 4.24) для рядка 5 таблиці Line2 постачальник повертає такий тип MIME (рис. 4.25).

```
content://com.example.trains/Line2/5
```

Рис. 4.24. **URI вмісту**

```
vnd.android.cursor.item/vnd.example.line2
```

Рис. 4.25. **Тип MIME для рядка 5 таблиці Line2**

У більшості постачальників контенту визначено константи класу-контракту для використовуваних у них типів MIME. Наприклад, клас-контракт `ContactsContract.RawContacts` постачальника контактів визначає константу `CONTENT_ITEM_TYPE` для типу MIME одного рядка необробленого контакту.

4.2. Створення постачальника контенту

Постачальник контенту управляє доступом до центрального сховища даних. Реалізація постачальника містить один чи кілька класів у додатку Android, а також елементи у файлі маніфесту. Один із класів реалізує підклас `ContentProvider`, який є інтерфейсом між постачальником та іншими додатками. Незважаючи на те, що постачальники контенту спочатку призначені для надання іншим програмам доступу до даних, у додатку, безсумнівно, можуть міститися операції, які дозволяють запитувати та змінювати дані, керовані постачальником.

4.2.1. Підготовка до створення постачальника

Перш ніж приступити до створення постачальника, слід виконати наведені далі дії:

1. **Вирішити, чи потрібен взагалі постачальник контенту.** Постачальник контенту потрібен у випадках, якщо необхідно реалізувати у своєму додатку одну або кілька таких функцій:

надання складних даних або файлів інших програм;

надання користувачам можливості копіювати складні дані з додатка в інші програми;

надання пошукових підказок, що налаштовують за допомогою платформи пошуку.

Для цього *не потрібен* постачальник для роботи з базою даних SQLite, якщо її планують використовувати виключно в додатку.

Якщо ще не прийняти остаточного рішення, слід ознайомитися зі статтею "*Основні відомості про постачальника контенту*", щоб дізнатися детальніше про постачальників контенту.

Після цього можна приступати до створення постачальника. Для цього слід виконати наведені далі дії:

1. Спроектувати базове сховище для своїх даних. Постачальник контенту надає дані двома способами:

Дані для файлів. Такі дані, які зазвичай надходять у файли, як фотографії, аудіо- чи відеодані. Файли слід зберігати в закритому просторі додатка. У відповідь на запит файла з іншої програми постачальник може запропонувати дескриптор файла.

Структуровані дані. Дані, які зазвичай надходять до бази даних, масиву або аналогічної структури. Дані слід зберігати в тій формі, яка сумісна з таблицями з рядків і стовпців. Рядок становить об'єкт, наприклад, користувача, позицію або облікову одиницю. Стовпець становить деякі дані про об'єкт, наприклад, назву користувача або вартість одиниці. Зазвичай дані такого типу зберігаються в базі даних SQLite, однак можна використовувати постійне сховище будь-якого типу.

2. Визначити конкретну реалізацію класу ContentProvider і його необхідні методи. Цей клас є інтерфейсом між даними та іншою частиною системи Android.

3. Визначити рядок центру постачальника, його URI контенту та стовпців. Якщо необхідно, щоб додаток постачальника обробляв наміри, також потрібно визначити дії намірів, додаткові дані та прапорці. Крім того, слід визначити дозволи, які будуть необхідні програмі для доступу до даних. Усі ці значення слід визначити як константи в окремому класі-контракті, надалі цей клас можна надати іншим розробникам.

4. Додати інші додаткові компоненти, наприклад, демонстраційні дані або реалізацію адаптера `AbstractThreadedSyncAdapter`, який слугує для синхронізації даних між постачальником і хмарним сховищем даних.

4.2.2. Проектування сховища даних

Постачальник контенту становить інтерфейс для передачі даних, збережених у структурованому форматі. Перш ніж створювати інтерфейс, слід визначити спосіб зберігання даних. Дані можна зберігати в будь-якій формі, а потім спроектувати інтерфейс для читання та запису даних за потреби.

В Android є деякі технології зберігання даних:

- у системі Android є API – бази даних SQLite, який використовується власними постачальниками Android для зберігання табличних даних. За допомогою класу `SQLiteOpenHelper` можна створювати бази даних, а клас `SQLiteDatabase` становить базовий клас для доступу до баз даних.

Зверніть увагу, що не обов'язково використовувати базу даних для реалізації свого репозиторія. Постачальник становить зовнішній набір таблиць, як у випадку з реляційною базою даних, однак це не є вимогою до внутрішньої реалізації постачальника;

- для зберігання файлів даних в Android передбачено різні API-інтерфейси для роботи з файлами. Якщо проектують постачальник, який пропонує такі мультимедійні дані, як музика чи відео, можна створити постачальник, що об'єднує табличні дані та файли;
- для роботи з мережевими даними використовуються класи `java.net` і `android.net`. Також можна синхронізувати мережеві дані з локальним сховищем даних (наприклад із базою даних), а потім надати такі дані у вигляді таблиць або файлів. Такий тип синхронізації продемонстровано на прикладі програми *адаптера синхронізації*.

Рекомендації щодо проектування даних

Ось декілька порад і рекомендацій щодо проектування структури даних постачальника:

у табличних даних завжди має бути стовпець для "основного ключа", який постачальник зберігає у вигляді унікального числового значення для кожного рядка. Можна використовувати це значення для пов'язування рядка з рядками в інших таблицях (використовуючи його як *ключ*). Незважаючи на те що можна використовувати будь-яку назву для цього стовпця, рекомендується вказати назву `BaseColumns._ID`, оскільки для пов'язування результатів запиту постачальника з `ListView` необхідно, щоб один з отримуваних стовпців називався `_ID`;

якщо заплановано надавати растрові зображення або дуже великі фрагменти даних для файлів, то дані слід зберігати у файлах, а потім надавати їх побічно, замість зберігання таких даних прямо в таблиці. У такому випадку необхідно повідомити користувачам постачальника про те, що для доступу до даних їм потрібно скористатися методом `ContentResolver`;

для зберігання даних різного розміру або з різною структурою слід використовувати тип BLOB. Наприклад, стовпець BLOB можна використовувати для зберігання *буфера протоколу або структури JSON*.

BLOB також можна використовувати для реалізації таблиці, що *не залежить від схеми*. У таблиці такого типу визначаються стовпець основного ключа, стовпець типу MIME та один або кілька спільних стовпців BLOB. На вміст даних у стовпцях BLOB вказує значення у стовпці типу MIME. Завдяки цьому в одній і тій же таблиці можна зберігати рядки різних типів. Прикладом таблиці, що не залежить від схеми, може бути таблиця з даними постачальника контенту `ContactsContract.Data`.

4.2.3. Проектування URI контенту

URI контенту становить URI, який визначає дані у постачальника. URI контенту можуть включати символічну назву всього постачальника (його **центр**) та назву, яка вказує на таблицю або файл (**шлях**). Додаткова частина URI з ідентифікатором указує на окремий рядок у таблиці. У кожного методу доступу до даних у класі `ContentProvider` є URI контенту (у вигляді аргументу), завдяки цьому можна визначити таблицю, рядок або файл для доступу.

Проектування центру постачальника

У постачальника, зазвичай, є тільки один центр, який є внутрішньою назвою у системі Android. Щоб уникнути конфліктів з іншими постачальниками основою центру постачальника мають виступати відомості про володіння доменом в Інтернеті (у зворотному порядку). Оскільки ця рекомендація також застосовується і до назв пакетів Android, можна визначити центр свого постачальника у вигляді розширення назви пакета, у якому міститься постачальник. Наприклад, якщо пакет Android називають `com.example.<appname>`, то центром постачальника має бути `com.example.<appname>.provider`.

Проектування структури шляху

Зазвичай, розробники створюють URI контенту на основі центру постачальника, додаючи до нього шлях, який указує на окремі таблиці. Наприклад, якщо є дві таблиці *table1* і *table2*, центр постачальника з попереднього прикладу слід об'єднати для формування таких URI контенту: `com.example.<appname>.provider/table1` і `com.example.<appname>.provider/table2`. Шляхи не обмежено одним сегментом, і не на кожному рівні шляху є таблиця.

Оброблення ідентифікаторів URI контенту

Зазвичай, постачальники надають доступ до одного рядка в таблиці шляхом прийняття URI контенту, у кінці якого вказано значення ідентифікатора рядка. Також постачальники перевіряють збіг значення ідентифікатора за стовпцем _ID в таблиці та надають запитуваний доступ до відповідного рядка.

Це спрощує створення загального методу проектування для програм, які отримують доступ до постачальника. Додаток відправляє запит постачальнику та відображає отриманий у результаті такого запиту об'єкт `Cursor` в об'єкті `ListView` за допомогою `CursorAdapter`. Для визначення `CursorAdapter` необхідно, щоб один із стовпців в об'єкті `Cursor` називався `_ID`.

Потім користувач вибирає в інтерфейсі один з рядків, щоб переглянути або змінити їх. Додаток отримує відповідний рядок з об'єкта `Cursor` у базовому об'єкті `ListView`, отримує значення `_ID` для цього рядка, додає його до URI контенту, а потім відправляє постачальнику запит на доступ. Потім постачальник може запросити або змінити рядок, обраний користувачем.

Шаблони URI контенту

Щоб допомогти у виборі дії для виконання зі вхідним URI контенту, в API постачальника є клас `UriMatcher`, який зіставляє шаблони URI контенту із цілочисельними значеннями. Такі цілочисельні значення можна використовувати в операторі `switch`, який вибирає відповідну дію для URI контенту, що відповідають певним шаблонам.

Для визначення збігу URI контенту із шаблоном використовують символи:

`*`: відповідність рядка будь-якої довжини з будь-якими припустимими символами;

`#`: відповідність рядка будь-якої довжини із цифрами.

Як приклад для проектування, написання коду для оброблення URI контенту рекомендовано використовувати центр постачальника `com.example.app.provider`, який розпізнає такі URI контенту, що вказують на таблиці:

content://com.example.app.provider/table1: таблиця `table1`;

content://com.example.app.provider/table2/dataset1: таблиця `dataset1`;

content://com.example.app.provider/table2/dataset2: таблиця `dataset2`;

content://com.example.app.provider/table3: таблиця `table3`.

Постачальник також розпізнає такі URI контенту, якщо до них додано ідентифікатор рядка (наприклад для рядка з ідентифікатором 1 у таблиці `table3.content://com.example.app.provider/table3/1`).

Можливе використання таких шаблонів URI контенту:

content://com.example.app.provider/*: збігається з будь-яким URI контенту у постачальника;

content://com.example.app.provider/table2/*: збігається з URI контенту в таблицях dataset1 і dataset2, однак не збігається з URI контенту в таблиці table1 або table3;

content://com.example.app.provider/table3/#: збігається з URI контенту для окремих рядків у таблиці `table3`, такими як `content://com.example.app.provider/table3/6` рядка з ідентифікатором `6`.

У фрагменті коду далі показано, як працюють методи у класі `UriMatcher`. Цей код обробляє URI для всієї таблиці інакше, ніж для URI окремого рядка, використовуючи шаблон URI вмісту `content://<authority>/<path>` для таблиць і шаблон `content://<authority>/<path>/<id>` – для окремих рядків.

Метод `addURI()` зіставляє центр постачальника та його шлях із цілочисельним значенням. Метод `match()` повертає ціле значення для URI. Оператор `switch` вибирає, треба йому виконати запит всієї таблиці або тільки окремого запису (рис. 4.26):

```
public class ExampleProvider extends ContentProvider {
...
    // Створюємо об'єкт UriMatcher.
    private static final UriMatcher sUriMatcher;
...
    /* Виклики функції addURI() знаходяться тут для всіх моделей URI
контенту, що слід розпізнати постачальнику. У цьому фрагменті коду пока-
зано виклик цієї функції лише до таблиці 3 */
...
    /* Установлюємо цілочисельне значення, що дорівнює 1, для кількох
рядків у таблиці 3 */
    sUriMatcher.addURI("com.example.app.provider", "table3", 1);

    /* Встановлюємо код для одного рядка, що дорівнює 2. У цьому випадку
     * використовується знак "#" знак.
```

Рис. 4.26. **Оброблення URI для всієї таблиці**

```
    * content://com.example.app.provider/table3/3" співпадає, але
     * "content://com.example.app.provider/table3 ні. */
    sUriMatcher.addURI("com.example.app.provider", "table3/#", 2);
...
    // Реалізуємо ContentProvider.query()
    public Cursor query(
        Uri uri,
        String[] projection,
        String selection,
        String[] selectionArgs,
        String sortOrder) {
...

      /* Вибираємо таблицю для запиту та порядок сортування, грунтуючись
на коді для вхідного URI. Тут також тільки вирази для таблиці 3. */
        switch (sUriMatcher.match(uri)) {
          // Якщо вхідний URI був для всієї таблиці
          case 1:
            if (TextUtils.isEmpty(sortOrder)) sortOrder = "_ID ASC";
                break;

          // Якщо вхідний URI був для одного рядка
          case 2:
            selection = selection + "_ID = " uri.getLastPathSegment();
            break;

          default:
            ...
           // Якщо URI не знайдено, ви маєте виконати оброблення помилок
        }
        // місце, щоб здійснити запит
    }
}
```

Закінчення рис. 4.26

Інший клас, `ContentUris`, надає зручні методи для роботи з частиною `id` URI контенту. Класи `Uri` і `Uri.Builder` містять зручні методи для синтаксичного аналізу наявних об'єктів Uri та створення нових.

4.2.4. Реалізація класу ContentProvider

Екземпляр класу `ContentProvider` управляє доступом до структурованого набору даних шляхом опрацювання запитів від інших додатків.

Отже, за всіх форм доступу викликається метод `ContentResolver`, який потім викликає конкретний метод `ContentProvider` для отримання доступу.

Необхідні методи

В абстрактному класі `ContentProvider` визначено шість абстрактних методів, які необхідно реалізувати в межах власного конкретного підкласу. Усі наведені далі методи, крім `onCreate()`, викликаються клієнтським додатком, що намагається отримати доступ до постачальника контенту.

`query()` – отримання даних від постачальника. Використовує аргументи для вибору таблиці для запиту, рядків і стовпців, які потрібно повернути, і вказання порядку сортування результатів. Повертає дані у вигляді об'єкта `Cursor`.

`insert()` – установлення рядка в постачальник. Використовує аргументи для вибору кінцевої таблиці та отримання значень стовпця, які слід використовувати. Повертає URI контенту для нового вставленого рядка.

`update()` – оновлення наявних рядків у постачальника. Використовує аргументи для вибору таблиці або рядків для оновлення, а також для отримання оновлених значень стовпця. Повертає кількість оновлених рядків.

`delete()` – видалення рядків із постачальника. Використовує аргументи для вибору таблиці або рядків для видалення. Повертає кількість видалених рядків.

`getType()` – повернення типу MIME, відповідного URI контенту.

`onCreate()` – ініціалізація постачальника. Система Android викликає цей метод відразу після створення вашого постачальника. Зверніть увагу, що постачальник не буде створено до тих пір, поки об'єкт `ContentResolver` не припинить спроби отримати доступ до нього.

Назва цих методів аналогічна підпису для ідентичних методів в об'єкті `ContentResolver`.

Під час реалізації цих методів слід ураховувати такі моменти.

Усі ці методи можна викликати відразу з декількох потоків, тому їх має бути реалізовано зі збереженням потокобезпеки, крім методу `onCreate()`.

Слід уникати занадто довгих операцій у методі `onCreate()`. Відкласти виконання завдань ініціалізації до тих пір, поки вони не будуть потрібні.

Незважаючи на те що мають реалізувати ці методи, код необов'язково має виконувати будь-які інші дії, крім повернення очікуваного типу даних. Наприклад, може знадобитися, щоб інші програми не мали можливості вставляти дані в деякі таблиці. Для цього можна ігнорувати виклик методу `insert()` і повернути `0`.

Реалізація методу `query()`

Метод `ContentProvider.query()` має повертати об'єкт `Cursor`, а під час перебоїв видавати `Exception`. Якщо як сховище даних використовувати базу даних SQLite, можна просто повернути об'єкт `Cursor`, що був повернутий одним із методів `query()` класу `SQLiteDatabase`. Якщо запит не відповідає ні одному рядку, слід повернути екземпляр об'єкта `Cursor`, метод `getCount()` повертає `0`. `null` слід повертати тільки в тому випадку, якщо під час оброблення запиту сталася внутрішня помилка.

Якщо не використовувати базу даних SQLite як сховище даних, слід звернутися до одного з конкретних підкласів об'єкта `Cursor`. Наприклад, клас `MatrixCursor` реалізує об'єкт Cursor, у якому кожен рядок становить масив класу `Object`. За допомогою цього класу потрібно скористатися методом `addRow()`, щоб додати новий рядок.

Слід пам'ятати, що система Android повинна мати можливість взаємодіяти з `Exception` у межах процесу. Система Android дозволяє це робити для зазначених далі винятків, які можуть бути корисні під час оброблення помилок запитів.

`IllegalArgumentException` (це виняток можна видати в разі, якщо постачальник отримує неправильний URI контенту);

`NullPointerException`.

Реалізація методу `insert()`

Метод `insert()` додає новий рядок у відповідний рядок, використовуючи значення в аргументі `ContentValues`. Якщо в аргументі `ContentValues` відсутня назва стовпця, можливо, знадобиться вказати для нього значення за замовчуванням (або в коді постачальника, або у схемі бази даних).

Цей метод має повертати URI контенту для нового рядка. Для цього слід додати значення `_ID` нового рядка (або інший основний ключ) до URI контенту таблиці, використовуючи метод `withAppendedId()`.

Реалізація методу `delete()`

Методом `delete()` необов'язково фактично видаляти рядки зі сховища даних. Якщо для роботи з постачальником використовувати адаптер синхронізації, слід розглянути можливість позначення віддаленого рядка прапорцем *delete*, замість остаточного видалення рядка. Адаптер синхронізації може перевірити наявність видалених рядків із прапорцем *delete*, щоб видалити їх із сервера перед видаленням із постачальника.

Реалізація методу `update()`

Метод `update()` приймає той же аргумент `ContentValues`, який застосовує метод `insert()`, і ті ж аргументи `selection` і `selectionArgs`, які використовуються методами `delete()` і `ContentProvider.query()`. Завдяки цьому код можна повторно використовувати між цими методами.

Реалізація методу `onCreate()`

Система Android викликає метод `onCreate()` під час запуску постачальника. У цьому методі слід виконувати тільки завдання ініціалізації, що швидко виконуються, а створення бази даних і завантаження їх відкласти до моменту фактичного отримання постачальником запиту на інформацію. Занадто довгі операції в методі `onCreate()` призводять до збільшення часу запуску постачальника. У свою чергу, це збільшує час відгуку постачальника на запити від інших додатків.

Наприклад, якщо використовувати базу даних SQLite, у методі `ContentProvider.onCreate()` можна створити новий об'єкт `SQLiteOpenHelper`, а потім створити таблиці SQL за першого відкриття бази даних. Щоб спростити це, за першого виклику методу `getWritableDatabase()` він автоматично викликає метод `SQLiteOpenHelper.onCreate()`.

У наведених далі фрагментах коду проілюстровано взаємодію між методами `ContentProvider.onCreate()` і `SQLiteOpenHelper.onCreate()`. У першому фрагменті коду наведено реалізацію методу `ContentProvider.onCreate()` (рис. 4.27):

```
public class ExampleProvider extends ContentProvider {

    /* Визначаємо дескриптор для допоміжного об'єкта бази даних. Клас
MainDatabaseHelper визначається у такому фрагменті. */
```

Рис. 4.27. **Взаємодія між методами `ContentProvider.onCreate()` і `SQLiteOpenHelper.onCreate()`**

```
    private MainDatabaseHelper mOpenHelper;
    // Визначаємо назву бази даних
    private static final String DBNAME = "mydb";

    // Зберігає об'єкт бази даних
    private SQLiteDatabase db;

    public boolean onCreate() {
     /* Створюємо новий допоміжний об'єкт. Цей спосіб завжди швидкий. */
      mOpenHelper = new MainDatabaseHelper(
            getContext(), // контекст додатка
            DBNAME,       // назва бази даних)
           null,         //використовуємо SQLite курсор за замовчуванням
            1             // номер версії
        );
        return true;
    }

    // Реалізуємо метод Insert постачальника
    public Cursor insert(Uri uri, ContentValues values) {
      /* Необхідно вставити тут код, щоб визначити, які таблиці відкри-
ті, обробити помилки, і так далі */

        ...

      /* Отримуємо записи бази даних.  */
      db = mOpenHelper.getWritableDatabase();
    }
}
```

Закінчення рис. 4.27

У наступному фрагменті коду наведено реалізацію методу `SQLiteOpenHelper.onCreate()`, включаючи допоміжний клас (рис. 4.28):

```
...
// Рядок, що визначає інструкцію SQL для створення таблиці
private static final String SQL_CREATE_MAIN = "CREATE TABLE " +
  "main " +              // Назва таблиці
  "(" +                 // Стовпці в таблиці
```

Рис. 4.28. **Реалізація методу `SQLiteOpenHelper.onCreate()`**

```
    " _ID INTEGER PRIMARY KEY, " +
    " WORD TEXT"
    " FREQUENCY INTEGER " +
    " LOCALE TEXT )";
...
/**
 * Клас Helper, який фактично створює та управляє основним сховищем даних
 * провайдера.
 */
protected static final class MainDatabaseHelper
                                        extends SQLiteOpenHelper {
  /**
   * Створює екземпляр відкритого помічника для SQLite сховища даних
   * постачальника. Не робити створення бази даних та оновлення тут.
   */
  MainDatabaseHelper(Context context) {
        super(context, DBNAME, null, 1);
    }

    /**
     * Створює сховище даних. Це викликається за спроби постачальника,
     * щоб відкрити сховище і SQLite повідомляє, що він не існує.
     */
    public void onCreate(SQLiteDatabase db) {

        // Створює основну таблицю
        db.execSQL(SQL_CREATE_MAIN);
    }
}
```

Закінчення рис. 4.28

4.2.5. Реалізація типів MIME постачальника контенту

У класі `ContentProvider` передбачено два методи для повернення типів MIME:

`getType()`.

Один із необхідних методів, який потрібно реалізувати для кожного постачальника.

`getStreamTypes()`.

Метод, який потрібно реалізувати в разі, якщо постачальник надає файли.

Типи MIME для таблиць

Метод `getType()` повертає об'єкт `String` у форматі MIME, який описує тип даних, що повертаються аргументом URI контенту. Аргумент `Uri` може бути шаблоном, а не у вигляді певного URI; у цьому випадку необхідно повернути тип даних, пов'язаний з URI контенту, який відповідає шаблону.

Для таких загальних типів даних, як текст, HTML або JPEG, метод `getType()` має повертати стандартний тип MIME. Повний список стандартних типів наведено на веб-сайті *IANA MIME Media Types*.

Для URI контенту, які вказують на одну або кілька рядків табличних даних, метод `getType()` має повертати тип MIME у форматі MIME постачальника, який є у системі Android:

частину типу: `vnd`;

частину підтипу:

– якщо шаблон URI призначено для одного рядка: `android.cursor.item/`;

– якщо шаблон URI призначений для декількох рядків: `android.cursor.dir/`;

частину постачальника: `vnd.<`*name*`>.<`*type*`>`.

Слід указати <*name*> й <*type*>. Значення <*name*> має бути унікальним глобально, а значення <*type*> має бути унікальним для відповідного шаблона URI. Як <*name*> рекомендовано використовувати назву компанії або частину назви пакета Android додатка. Як <*type*> рекомендовано використовувати рядок, що визначає пов'язану з URI таблицю.

Наприклад, якщо центр постачальника є `com.example.app.provider`, що надає таблицю `table1`, то тип MIME для кількох рядків у таблиці `table1` буде таким (рис. 4.29):

```
vnd.android.cursor.dir/vnd.com.example.provider.table1
```

Рис. 4.29. **Тип MIME для кількох рядків у таблиці `table1`**

Для одного рядка в таблиці `table1` тип MIME буде таким (рис. 4.30):

```
vnd.android.cursor.item/vnd.com.example.provider.table1
```

Рис. 4.30. **Тип MIME для одного рядка в таблиці `table1`**

Типи MIME для файлів

Якщо постачальник надає файли, необхідно реалізувати метод `getStreamTypes()`. Цей метод повертає масив `String` із типами MIME для файлів, що повертаються постачальником для заданого URI контенту. Запропоновані постачальником типи слід сортувати за допомогою аргументу фільтра типів MIME, щоб поверталися тільки ті типи MIME, які необхідно обробити клієнту.

Наприклад, слід розглянути постачальника, який надає фотографії у вигляді файлів у форматах `.jpg`, `.png` і `.gif`. Якщо додаток викликає метод `ContentResolver.getStreamTypes()` із рядком фільтра `image/*` (щось на кшталт "зображення"), то метод `ContentProvider.getStreamTypes()` має повертати такий масив (рис. 4.31):

```
{ "image/jpeg", "image/png", "image/gif"}
```

Рис. 4.31. **Масив**

Якщо ж додатку потрібні тільки файли `.jpg`, то він викликає метод `ContentResolver.getStreamTypes()` із рядком фільтра `*\/jpeg;` метод `ContentProvider.getStreamTypes()` одночасно має повертати таке (рис. 4.32):

```
{"image/jpeg"}
```

Рис. 4.32. **Дані, що повертає метод `ContentProvider.getStreamTypes()`**

Якщо постачальник не надає жоден із типів MIME, запитаних у рядку фільтра, то метод `getStreamTypes()` має повертати `null`.

4.2.6. Реалізація класу-контракту

Клас-контракт становить клас `public final`, у якому містяться визначення констант для URI, назв стовпців, типів MIME та інших метаданих постачальника. Клас установлює контрактні відносини між постачальником та іншими додатками шляхом забезпечення прямого доступу до постачальника, навіть у разі зміни фактичних значень URI, назв стовпців і т.д.

Клас-контракт також корисний для розробників тим, що в ньому містяться мнемонічні назви для його констант, завдяки чому знижується

ризик того, що розробники скористаються неправильними значеннями для назв стовпців або URI. Оскільки це клас, він може містити документацію Javadoc. Інтегровані середовища розроблення, такі як Eclipse, можуть автоматично заповнювати назви констант із класу-контракту та відображати Javadoc для констант.

У розробників немає доступу до файла класу-контракту з додатка, проте вони можуть статично скомпілювати клас-контракт у свій додаток із наданого файла `.jar`.

Прикладом класу-контракту може бути клас `ContactsContract` і його вкладені класи.

4.2.7. Реалізація дозволів постачальника контенту

Далі наведено короткий огляд основних моментів:

за замовчуванням файли з даними зберігаються у внутрішньому сховищі пристрою й доступні тільки додатку та постачальнику;

створювані бази даних `SQLiteDatabase` також доступні тільки додатку та постачальнику;

файли з даними, які зберігають у зовнішньому сховищі, за замовчуванням є *загальнодоступними, які може зчитати будь-який користувач*. Не вдасться використовувати постачальника контенту для обмеження доступу до файлів, які зберігають у зовнішньому сховищі, оскільки додатки можуть використовувати інші виклики API для їхнього читання або запису;

виклики методу для відкриття або створення файлів або баз даних SQLite, що знаходяться у внутрішньому сховищі на вашому пристрої, потенційно можуть надати всі інші додатки для доступу як на запис, так і на читання даних. Якщо використовувати внутрішній файл або базу даних як сховище постачальника, виконати читання або запис даних у якому може будь-який користувач, то дозволів, заданих у маніфесті постачальника, буде явно недостатньо для захисту даних. За замовчуванням доступ до файлів і баз даних у внутрішньому сховищі є закритим, і не слід змінювати параметри доступу до сховища постачальника.

За потреби слід використовувати *дозволи постачальника контенту,* для управління доступом до даних, дані слід зберігати у внутрішніх файлах, у базах даних SQLite або хмарі (наприклад, на віддаленому сервері), а доступ до файлів і баз даних має бути надано тільки додатку.

Реалізація дозволів

Будь-який додаток може виконувати читання даних у постачальнику або записувати їх, навіть якщо відповідні дані є закритими, оскільки за замовчуванням для постачальника не задано дозволи. Щоб змінити ці налаштування,слід задати дозволи для постачальника у файлі маніфесту за допомогою атрибутів елемента `<provider>` або його дочірніх елементів. Можна задати дозволи, які застосовують до всього постачальника, або тільки до певних таблиць, або навіть тільки до певних записів, або всьому дереву.

Для задавання дозволів використовують один або кілька елементів `<permission>` у файлі маніфесту. Щоб дозволи були унікальними для постачальника, слід використовувати області, аналогічні Java, для атрибута `android:name`. Наприклад, надати дозволу на читання назву `com.example.app.provider.permission.READ_PROVIDER`.

Далі перелічено області дозволів для постачальника, починаючи з дозволів, які застосовують до всього постачальника, і закінчуючи більш детальними дозволами. Більш докладні дозволи мають перевагу над дозволами з більш широкими областями:

одиничний дозвіл на читання/запис на рівні постачальника;

одиничний дозвіл, який управляє доступом як на читання, так і на запис для всього постачальника, який задають за допомогою атрибута `android:permission` елемента `<provider>`;

окремий дозвіл на читання/запис на рівні постачальника;

дозвіл на читання та запис для всього постачальника. Такі дозволи задають за допомогою атрибутів `android:readPermission` й `android:writePermission` елемента `<provider>`. Вони мають переважну силу над дозволом, заданим за допомогою атрибута `android:permission`;

дозвіл на рівні шляху;

дозвіл на читання, запис або читання/запис для URI контенту в постачальника. Кожен URI, що підлягає управлінню, задають за допомогою дочірнього елемента `<path-permission>` елемента `<provider>`. Для кожного зазначеного раніше URI контенту можна задати дозвіл на читання/запис, тільки читання або тільки запис, або всі три дозволи. Дозволи на читання та запис мають переважну силу над дозволом на читання/запис. Крім того, дозволи на рівні шляху мають переважну силу над дозволами на рівні постачальника;

тимчасовий дозвіл. Дозволи цього рівня надають додатку тимчасовий доступ, навіть якщо у додатка немає дозволів, які, зазвичай, потрібні.

Функція тимчасового доступу обмежує набір дозволів, які з додатком необхідно запросити у своєму маніфесті. Якщо включено тимчасові дозволи, єдиними додатками, яким потрібні "постійні" дозволи на роботу з постачальником, є ті, які безперервно отримують доступ до всіх даних.

Слід розглянути приклад з дозволами, які необхідно реалізувати для постачальника електронної пошти та додатка, коли вам необхідно дозволити зовнішньому додатку для перегляду зображень відображати вкладені в листи фотографії з постачальника. Щоб надати засобу перегляду зображень необхідний доступ без запиту дозволів, слід задати тимчасові дозволи для URI контенту фотографій. Спроектувати додаток для роботи з електронною поштою таким чином, щоб у випадках, коли користувач бажає відобразити фотографію, додаток відправляв би намір, у якому міститься URI контенту фотографії та прапорці дозволу для засобу перегляду зображень. Потім засіб перегляду зображень може надсилати операторові електронної пошти запит на отримання фотографії, навіть якщо в засобу перегляду відсутній звичайний дозвіл на читання даних із постачальника.

Щоб включити тимчасові дозволи, слід задати атрибут `android:grantUriPermissions` для елемента `<provider>` або додати один чи кілька дочірніх елементів `<grant-uri-permission>` в елемент `<provider>`. Якщо використовувати тимчасові дозволи, необхідно викликати метод `Context.revokeUriPermission()` кожен раз, коли здійснюють віддалену підтримку URI контенту з постачальника, а URI контенту пов'язано з тимчасовим дозволом.

Значення атрибута визначає, яка частина постачальника доступна. Якщо для атрибута задано значення true, система надасть тимчасові дозволи для всього постачальника, скасовуючи тим самим будь-які інші дозволи, потрібні на рівні постачальника або на рівні шляху.

Якщо для прапорця задано значення false, необхідно додати дочірні елементи `<grant-uri-permission>` у свій елемент `<provider>`. Кожен дочірній елемент задає URI контенту, для яких надано тимчасовий дозвіл.

Щоб делегувати додатку тимчасовий доступ, у намірі має бути вказано прапорці `FLAG_GRANT_READ_URI_PERMISSION`, або `FLAG_GRANT_WRITE_URI_PERMISSION`, або обидва прапора. Ці прапорці задають за допомогою методу `setFlags()`.

Якщо атрибут `android:grantUriPermissions` відсутній, передбачають, що його значення false.

4.2.8. Елемент `<provider>`

Як і компоненти `Activity` й `Service`, підклас класу `ContentProvider` має бути визначено у файлі маніфесту програми за допомогою елемента `<provider>`. Далі вказано інформацію, яку система Android отримує із цього елемента:

1. Центр постачальника (`android:authorities`).
2. Символічні назви, які ідентифікують увесь постачальник у системі.
3. Назва класу постачальника (`android:name`).
4. Клас, який реалізує клас `ContentProvider`.
5. Дозволи.
6. Атрибути, які визначають дозволи, необхідні іншим додаткам для доступу до даних у постачальника:

`android:grantUriPermissions` – прапорець тимчасового дозволу;

`android:permission` – одиничний дозвіл на читання/запис на рівні постачальника;

`android:readPermission` – дозвіл на читання на рівні постачальника;

`android:writePermission` – дозвіл на запис на рівні постачальника.

7. Атрибути запуску та управління.

Наступні атрибути визначають порядок і час запуску постачальника системи Android, характеристики процесу постачальника, а також інші параметри виконання:

`android:enabled` – прапорець, що дозволяє системі запускати постачальника;

`android:exported` – прапорець, що дозволяє іншим програмам використовувати цей постачальник;

`android:initOrder` – порядок запуску постачальника (щодо інших постачальників в одному й тому процесі);

`android:multiProcess` – прапорець, що дозволяє системі запускати постачальника в тому ж процесі, що й викликає клієнт;

`android:process` – назва процесу, у якому запускається постачальник;

`android:syncable` – прапорець, який указує на те, що дані в постачальнику слід синхронізувати з даними на сервері.

8. Інформаційні атрибути.
9. Додатковий значок і мітка для постачальника:

`android:icon` – графічний ресурс, що містить значок для постачальника. Значок відображається поруч із міткою постачальника у списку додатків у розділі *Налаштування> Програми> Все*.

`android:label` – інформаційна мітка з описом постачальника, або його даних, або обидва описи. Мітка відображається у списку додатків у розділі *Налаштування> Програми> Все*.

4.2.9. Наміри та доступ до даних

Додатки можуть отримувати доступ до постачальника контенту, оминаючи за допомогою об'єктів `Intent`. Додаток до того ж не викликає будь-які методи класів `ContentResolver` чи `ContentProvider`. Замість цього він відправляє намір, що запускає операцію, яка, зазвичай, є частиною власного додатка постачальника. Отримання та відображення даних у своєму інтерфейсі виконує кінцева операція. Залежно від дії, зазначеної в намірі, кінцева операція також може запропонувати користувачеві внести зміни в дані постачальника. У намірі також можуть міститися додаткові дані, які кінцева операція відображає в інтерфейсі; потім користувачеві надають можливість змінити ці дані, перш ніж використовувати їх для зміни даних у постачальника.

Можливо, доступ за допомогою наміру потрібно використовувати для забезпечення цілісності даних. Для вставлення, оновлення та видалення даних у постачальника може існувати точно визначений програмний код, який реалізує його функціональні можливості. У цьому випадку надання іншим додаткам прямого доступу для зміни даних може призвести до того, що інформація буде недійсною. Якщо потрібно надати розробникам можливість доступу за допомогою намірів, слід ретельно задокументувати таку функцію. Пояснити їм, чому доступ за допомогою намірів через інтерфейс користувача програми є набагато кращим від зміни даних за допомогою їхнього коду.

Оброблення вхідного наміру для зміни даних постачальника нічим не відрізняється від оброблення інших намірів.

4.3. Завантажувачі (`Loaders`)

Завантажувачі, які з'явилися в Android 3.0, спрощують асинхронне завантаження даних в операцію або фрагмент. Завантажувачі мають такі властивості:

вони є для будь-яких операцій Activity та фрагментів Fragment;

вони забезпечують асинхронне завантаження даних;

вони відстежують джерело своїх даних і видають нові результати за зміни контенту;

вони можуть автоматично перепідключатися до останнього курсора завантажувача під час відтворення після зміни конфігурації. Таким чином, їм не потрібно повторно запитувати свої дані.

4.3.1. Зведена інформація про API-інтерфейс завантажувача

Є кілька класів та інтерфейсів, які можуть використовувати завантажувачі в додатку. Їх наведено в табл. 4.3:

Таблиця 4.3

Властивості завантажувачів

Класи / інтерфейси	Описи
`LoaderManager`	Абстрактний клас, що пов'язується з `Activity` чи `Fragment` для управління одним або декількома інтерфейсами `Loader`. Це дозволяє додатку управляти операціями, що виконуються досить довго разом із життєвим циклом `Activity` чи `Fragment`; найчастіше цей клас використовують із `CursorLoader`, однак додатку можуть писати свої власні завантажувачі для роботи з іншими типами даних. Є тільки один клас `LoaderManager` на операцію або фрагмент. Однак у класі `LoaderManager` може бути кілька завантажувачів
`LoaderManager.LoaderCallbacks`	Інтерфейс зворотного виклику, що забезпечує взаємодію клієнта з `LoaderManager`. Наприклад, за допомогою методу зворотного виклику `onCreateLoader()` створюється новий завантажувач
`Loader`	Абстрактний клас, який виконує асинхронне завантаження даних. Це базовий клас для завантажувача. Зазвичай використовують `CursorLoader`, але можна реалізувати і власний підклас. Коли завантажувачі активні, вони мають відстежувати джерело своїх даних і видавати нові результати за зміни контенту
`AsyncTaskLoader`	Абстрактний завантажувач, який надає `AsyncTask` для виконання роботи
`CursorLoader`	Підклас класу `AsyncTaskLoader`, який запитує `ContentResolver` і повертає `Cursor`. Цей клас реалізує протокол `Loader` стандартним способом для виконання запитів до курсора. Його побудовано на `AsyncTaskLoader` для виконання запиту до курсора у фоновому потоці, щоб не блокувати інтерфейс користувача програми. Використання цього завантажувача – це найкращий спосіб асинхронного завантаження даних із `ContentProvider`, замість виконання керованого запиту через платформу або API-інтерфейси операції

Наведені в цій таблиці класи та інтерфейси є найбільш важливими компонентами, за допомогою яких у додатку реалізується завантажувач. Під час створення кожного завантажувача не потрібно використовувати всі ці компоненти, проте завжди слід указувати посилання на `LoaderManager` для ініціалізації завантажувача та використовувати реалізацію класу `Loader`, наприклад `CursorLoader`.

4.3.2. Використання завантажувачів у додатку

У цьому підрозділі описано використання завантажувачів у додатку для Android. У додатках, що використовують завантажувачі, зазвичай є такі елементи:

`Activity` чи `Fragment`;

екземпляр `LoaderManager`;

`CursorLoader` для завантаження даних, які видаються `ContentProvider`. Також можна реалізувати власний підклас класу `Loader` чи `AsyncTaskLoader` для завантаження даних з іншого джерела;

реалізація для `LoaderManager.LoaderCallbacks`. Саме тут створюють нові завантажувачі та ведуть управління посиланнями на наявні завантажувачі;

спосіб відображення даних завантажувача, наприклад `SimpleCursorAdapter`;

джерело даних, наприклад `ContentProvider`, коли використовують `CursorLoader`.

Запуск завантажувача. `LoaderManager` управляє одним або декількома екземплярами `Loader` в `Activity` чи `Fragment`. Є тільки один `LoaderManager` на кожну операцію або кожен фрагмент.

`Loader` зазвичай ініціалізується в методі операції `onCreate()` або методі фрагмента `onActivityCreated()`. Роблять це в такий спосіб (рис. 4.33):

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

Рис. 4.33. **Ініціалізація `Loader`**

Метод `initLoader()` приймає такі параметри:

унікальний ідентифікатор, що позначає завантажувач. У цьому прикладі ідентифікатором є `0`;

необов'язкові аргументи, які передаються завантажувачу під час побудови (у цьому прикладі це `null`);

реалізація `LoaderManager.LoaderCallbacks`, яка викликає клас `LoaderManager` для видачі подій завантажувача. У цьому прикладі локальний клас реалізує інтерфейс `LoaderManager.LoaderCallbacks`, тому він передає посилання самому собі: `this`.

Виклик `initLoader()` забезпечує ініціалізацію завантажувача. Можливий один із двох результатів:

1. Якщо завантажувач, указаний за допомогою ідентифікатора, вже існує, буде повторно використано завантажувач, створений останнім.

2. Якщо завантажувач, указаний за допомогою ідентифікатора, не існує, `initLoader()` викликає метод `LoaderManager.LoaderCallbacks` з `onCreateLoader()`. Саме тут реалізується код для створення екземпляра та повернення нового завантажувача.

У будь-якому випадку ця реалізація `LoaderManager.LoaderCallbacks` пов'язується із завантажувачем і буде викликатися за зміни стану завантажувача. Якщо в момент цього виклику викликається компонент, то він перебуває в запущеному стані, це означає, що запитаний завантажувач вже існує і сформував свої дані. У цьому випадку система відразу ж викличе `onLoadFinished()` (під час `initLoader()`), слід бути готовим до цього.

Слід звернути увагу, що метод `initLoader()` повертає створюваний клас `Loader`, але записувати посилання на нього не потрібно. Клас `LoaderManager` управляє життєвим циклом завантажувача автоматично. Клас `LoaderManager` почне завантаження та закінчує його за потреби, а також підтримує стан завантажувача і пов'язаного з ним контенту. А це означає, що будуть рідко взаємодіяти із завантажувачами безпосередньо (проте приклад використання методів завантажувача для тонкого налаштування його поведінки див. у зразку коду `LoaderThrottle`). Для втручання у процес завантаження під час виникнення певних подій зазвичай використовують методи `LoaderManager i LoaderCallbacks`.

Перезапуск завантажувача. Під час використання методу `initLoader()`, як показано раніше, він задіює наявний завантажувач із зазначеним ідентифікатором, у разі його наявності. Якщо такого завантажувача немає, метод його створить. Однак іноді старі дані потрібно відкинути та почати все заново.

Для видалення старих даних використовують метод `restartLoader()`. Наприклад, ця реалізація методу `SearchView.OnQueryTextListener` перезапускає завантажувач, коли змінюється запит користувача. Завантажувач необхідно перезавантажити, із тим щоб він міг використовувати змінений фільтр пошуку для виконання нового запиту (рис. 4.34):

```
public boolean onQueryTextChanged(String newText) {
    // Called when the action bar search text has changed.  Update
    // the search filter, and restart the loader to do a new query
    // with this filter.
    mCurFilter = !TextUtils.isEmpty(newText) ? newText : null;
    getLoaderManager().restartLoader(0, null, this);
    return true;
}
```

Рис. 4.34. **Реалізація методу `SearchView.OnQueryTextListener`**

Використання зворотних викликів класу `LoaderManager`. `LoaderManager.LoaderCallbacks` становить інтерфейс зворотного виклику, який дозволяє клієнту взаємодіяти із класом `LoaderManager`.

Очікують, що завантажувачі, зокрема `CursorLoader`, будуть зберігати свої дані після їхньої зупинки. Це дозволяє додаткам зберігати свої дані в методах `onStop()` та `onStart()` операції або фрагмента, із тим щоб, коли користувач повернеться в додаток, йому не довелося чекати, поки дані завантажаться заново. Методи `LoaderManager.LoaderCallbacks` використовують, щоб дізнатися, коли потрібно створити новий завантажувач, а також для того, щоб указати додаткам, коли прийшов час перестати використовувати дані завантажувача.

Інтерфейс `LoaderManager.LoaderCallbacks` використовує такі методи:

`onCreateLoader()` – створює екземпляр і повертає новий клас `Loader` для цього ідентифікатора;

`onLoadFinished()` – викликається, коли створений раніше завантажувач завершив завантаження;

`onLoaderReset()` – викликається, коли стан створеного раніше завантажувача скидається, у результаті чого його дані губляться.

4.3.3. onCreateLoader

За спроби доступу до завантажувача (наприклад, за допомогою методу `initLoader()`), він перевіряє, чи існує завантажувач, указаний

за допомогою ідентифікатора. Якщо той не існує, він викликає метод `LoaderManager.LoaderCallbacks.onCreateLoader()`. Саме тут і створюється новий завантажувач. Зазвичай, це буде клас `CursorLoader`, однак можна реалізувати і власний підклас класу `Loader`.

У цьому прикладі метод зворотного виклику `onCreateLoader()` створює клас `CursorLoader` (рис. 4.35). Треба побудувати клас `CursorLoader` за допомогою його методу конструктора, для чого потрібен повний набір інформації, необхідної для виконання запиту до `ContentProvider`. Зокрема, потрібно:

`uri` – URI контенту, який необхідно отримати;

`projection` – список стовпців, які буде повернуто. Під час передачі `null` буде повернуто всі стовпці, а це неефективно;

`selection` – фільтр, що оголошує, які рядки повертати. Він відформатований у вигляді пропозиції SQL WHERE (за винятком самого WHERE). Під час передачі `null` буде повернуто всі рядки для цього URI;

`selectionArgs` – у вибірку можна включити символи "`?`", які буде замінено значеннями з `selectionArgs` у порядку проходження у вибірці. Значення буде прив'язано як рядки;

`sortOrder` – порядок розташування рядків, відформатований у вигляді пропозиції SQL ORDER BY (за винятком самого ORDER BY). Під час передачі `null` будуть використовувати стандартний порядок сортування, тому список, можливо, буде несортованими.

Наприклад:

```
// Якщо не нуль, це поточний фільтр, який надав користувач.
String mCurFilter;
...
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    //  Це викликається, коли потрібно створити новий Loader. Цей зразок
має лише
    // один Loader, тому ми не дбаємо про ідентифікатор.
    // Спочатку виберіть основний URI для використання
    // залежно від того, чи в даний час ми фільтруємо.
    Uri baseUri;
    if (mCurFilter != null) {
        baseUri = Uri.withAppendedPath(Contacts.CONTENT_FILTER_URI,
                  Uri.encode(mCurFilter));
```

Рис. 4.35. **Метод зворотного виклику `onCreateLoader()`**

```
    } else {
        baseUri = Contacts.CONTENT_URI;
    }

    // Тепер створіть і поверніть CursorLoader, який буде стежити за
створенням курсору для відображуваних даних.
    String select = "((" + Contacts.DISPLAY_NAME + " NOTNULL) AND ("
            + Contacts.HAS_PHONE_NUMBER + "=1) AND ("
            + Contacts.DISPLAY_NAME + " != '' ))";
    return new CursorLoader(getActivity(), baseUri,
            CONTACTS_SUMMARY_PROJECTION, select, null,
            Contacts.DISPLAY_NAME + " COLLATE LOCALIZED ASC");
}
```

Закінчення рис. 4.35

4.3.4. Метод onLoadFinished

Цей метод викликається, коли створений раніше завантажувач завершив завантаження. Цей метод гарантовано викликається до вивільнення останніх даних, які були надані цьому завантажувачу. До цього моменту необхідно повністю перестати використовувати старі дані (оскільки їх скоро буде замінено). Однак цього не слід робити самостійно, оскільки даними володіє завантажувач і він подбає про це.

Завантажувач вивільнить дані, як тільки дізнається, що додаток їх більше не використовує. Наприклад, якщо даними є курсор із `CursorLoader`, не слід викликати `close()` самостійно. Якщо курсор розміщується в `CursorAdapter`, слід використовувати метод `swapCursor()` із тим, щоб старий `Cursor` не закрився. Наприклад (рис. 4.36):

```
// Цей адаптер використовується для відображення даних списку.
SimpleCursorAdapter mAdapter;
...

public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
  // Помістіть курсор у новому вікні. (Після того як ми повернемося,
система буде закривати старий курсор.)
    mAdapter.swapCursor(data);
}
```

Рис. 4.36. **Реалізація методу** `onLoadFinished`

4.3.5. Метод onLoaderReset

Цей метод викликається, коли стан створеного раніше завантажувача скидається, у результаті чого його дані губляться. Цей зворотний виклик дозволяє дізнатися, коли дані ось-ось буде вивільнено, із тим щоб можна було видалити своє посилання на них.

Ця реалізація викликає `swapCursor()` зі значенням `null` (рис. 4.37):

```
// Адаптор використовується для візуалізації даних списку.
SimpleCursorAdapter mAdapter;
...
public void onLoaderReset(Loader<Cursor> loader) {
    // Це визивається, коли останній курсор, наданий на onLoadFinished()
приблизно буде закритий. Ми повинні переконатися, що ми більше
не використовуємо його.
    mAdapter.swapCursor(null);
}
```

Рис. 4.37. **Реалізація методу** `onLoadReset`

4.4. Постачальник контактів

Постачальник контактів становить ефективний і гнучкий компонент Android, який управляє центральним репозиторієм пристрою, у якому зберігаються призначені для користувача дані. Постачальник контактів – це джерело даних, які відображаються в меню "Контакти" на пристрої. Також можна отримати доступ до цих даних у своєму власному додатку та організувати обмін такими даними між пристроєм і службами в Інтернеті. Постачальник взаємодіє із широким набором джерел даних і намагається організувати управління якомога більшою кількістю даних про кожну людину, тому організація постачальника досить складна. Із цієї причини API постачальника містить широкий набір класів-контрактів та інтерфейси, що відповідають як за отримання даних, так і за їхню зміну.

У цьому підрозділі розглядають такі питання:

основна структура постачальника;

порядок отримання даних від постачальника;

порядок зміни даних у постачальнику;

порядок запису адаптера синхронізації для синхронізації даних, отриманих із сервера, із даними в постачальнику контактів.

Під час вивчення цього матеріалу мають на увазі, що студенти вже знайомі з основами постачальників контенту Android. Додаткові відомості про постачальників контенту Android подані у керівництві *"Основні відомості про постачальника контенту". Адаптер синхронізації* є прикладом використання такого додатка для обміну даними між постачальником контактів і додатком, розміщеним у веб-службах Google.

4.4.1. Структура постачальника контактів

Постачальник контактів становить постачальник контенту Android. Він містить три типи даних про користувача, кожен із яких вказано в окремій таблиці, що надається постачальником, як показано на рис. 4.38.

Як назви цих трьох таблиць, зазвичай, використовують назви відповідних класів-контрактів. Ці класи визначають константи для URI контенту, назви стовпців і значень у стовпцях цих таблиць:

таблиця `ContactsContract.Contacts`. Рядки в цій таблиці містять дані про різних користувачів, отримані шляхом агрегації рядків необроблених контактів;

таблиця `ContactsContract.RawContacts`. Рядки в цій таблиці містять зведені дані про користувача, що належать до акаунту користувача та його типу;

таблиця `ContactsContract.Data`. Рядки в цій таблиці містять відомості про необроблені контакти, такі як адреса електронної пошти або номери телефонів.

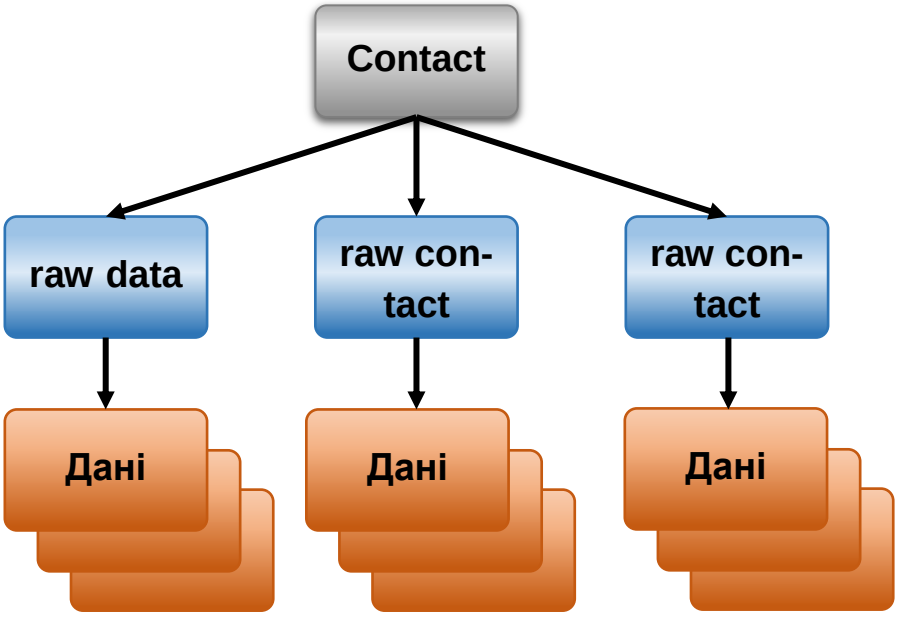

Рис. 4.38. **Структура таблиці постачальника контактів**

Інші таблиці, подані класами-контрактами в `ContactsContract`, становлять допоміжні таблиці, які постачальник контактів використовує для управління своїми операціями або для підтримки певних функцій, наявних у додатку пристрою *Контакти* або додатках для телефонного зв'язку.

4.4.2. Необроблені контакти

Необроблений контакт становить дані про користувача, що надходять з одного акаунту певного типу. Оскільки джерелом даних про користувача в постачальнику контактів може бути відразу кілька онлайн-служб, для одного й того ж користувача в постачальнику контактів може існувати кілька необроблених контактів. Це також дозволяє користувачеві об'єднувати призначені для користувача дані з декількох акаунтів одного й того ж типу.

Більша частина даних необробленого контакту не зберігається в таблиці `ContactsContract.RawContacts`. Замість цього вони зберігаються в одному або декількох рядках в таблиці `ContactsContract.Data`. У кожному рядку даних є стовпець `Data.RAW_CONTACT_ID`, у якому містяться значення `android.provider.BaseColumns#_ID RawContacts._ID` його батьківського рядка `ContactsContract.RawContacts`.

Важливі стовпці необроблених контактів

У табл. 4.4 вказано стовпці таблиці `ContactsContract.RawContacts`, які мають велике значення. Треба обов'язково ознайомитися із примітками, наведеними після цієї таблиці.

Таблиця 4.4

Важливі стовпці необроблених контактів

Назви стовпців	Використання	Примітки
1	2	3
`ACCOUNT_NAME`	Назва облікового запису для типу акаунту, який є джерелом даних для необробленого контакту. Наприклад, назва облікового запису Google є однією з електронних адрес пошти Gmail власника пристрою. Додаткові відомості дивіться у наступному записі `ACCOUNT_TYPE`	Формат цієї назви залежить від типу облікового запису. Це не обов'язково адреса електронної пошти

Закінчення табл. 4.4

1	2	3
ACCOUNT_TYPE	Тип акаунту, який є джерелом даних для необробленого контакту. Наприклад, тип акаунту Google – com.google. Завжди слід указувати тип акаунту та ідентифікатор домену, яким ви володієте або управляєте. Це дозволить гарантувати унікальність типу облікового запису	У типу акаунту, який є джерелом даних контактів, зазвичай, є пов'язаний із ним адаптер синхронізації, який синхронізує дані з постачальником контактів
DELETED	Прапорець *deleted* для необробленого контакту	Цей прапорець дозволяє постачальнику контактів зберігати рядок усередині себе до тих пір, поки адаптери синхронізації не зможуть видалити цей рядок із серверів, а потім видалити його зі сховищ

Примітки. Далі наведено важливі примітки до таблиці ContactsContract.RawContacts:

назва необробленого контакту не зберігається у його рядку в таблиці ContactsContract.RawContacts. Замість цього вона зберігається в таблиці ContactsContract.Data у рядку ContactsContract.CommonDataKinds.Struc turedName. У необробленого контакту є в таблиці ContactsContract.Data тільки один рядок такого типу.

Увага! Щоб використовувати в рядку необробленого контакту дані власного облікового запису, рядок спочатку необхідно зареєструвати його в класі AccountManager. Для цього слід запропонувати користувачам додати тип акаунту та його назву в список акаунтів. Якщо не зробити цього, постачальник контактів автоматично видалить рядок необробленого контакту.

Наприклад, якщо необхідно, щоб у додатку зберігалися дані контактів для веб-служби в домені com.example.dataservice, і акаунт користувача служби має такий вигляд: becky.sharp@dataservice.example.com, то користувачеві спочатку необхідно додати "тип" акаунту (com.example.dataservice) і його "назву" (becky.smart@dataservice.example.com) перш ніж додаток зможе додавати рядки необроблених контактів. Цю вимогу можна вказати в документації для користувачів, а також можна запропонувати

користувачеві додати тип та назву свого облікового запису або реалізувати обидва ці варіанти.

Джерела даних необроблених контактів. Щоб зрозуміти, що таке необроблений контакт, слід розглянути приклад із користувачем Emily Dickinson, на пристрої якої є три таких акаунти:

`emily.dickinson@gmail.com`;

`emilyd@gmail.com`;

belle_of_amherst у Twitter.

Вона включила функцію *Синхронізувати контакти* для всіх трьох цих акаунтів у налаштуваннях *Акаунти*.

Припустім, що Emily Dickinson відкриває браузер, входить у Gmail під назвою `emily.dickinson@gmail.com`, потім відкриває *Контакти* і додає новий контакт Thomas Higginson. Пізніше вона знову входить у Gmail під назвою `emilyd@gmail.com` і відправляє листа користувачеві Thomas Higginson, який автоматично додається у її контакти. Також вона підписана на новини від colonel_tom (акаунт користувача Thomas Higginson у Twitter) у Twitter. У результаті цих дій постачальник контактів створює три необроблених контакти:

1. Необроблений контакт Thomas Higginson, пов'язаний з акаунтом emily.dickinson@gmail.com. Тип цього акаунту – Google.

2. Другий необроблений контакт Thomas Higginson, пов'язаний із акаунтом `emilyd@gmail.com`. Тип цього акаунту – також Google. Другий необроблений контакт створюється навіть у тому випадку, якщо його назва повністю збігається з назвою попереднього контакту, оскільки перший було додано для іншого облікового запису.

3. Третій необроблений контакт Thomas Higginson, пов'язаний з акаунтом *belle_of_amherst*. Тип цього акаунту – Twitter.

4.4.3. Дані

Як уже зазначалося раніше, дані необробленого контакту зберігаються в рядку `ContactsContract.Data`, який пов'язано зі значенням _ID необробленого контакту. Завдяки цьому для одного необробленого контакту може існувати декілька примірників одного й того ж типу даних (наприклад, адреса електронної пошти або номерів телефонів). Наприклад, якщо в контакті Thomas Higginson для акаунту `emilyd@gmail.com` (рядок необробленого контакту Thomas Higginson, пов'язаний з обліковим записом Google `emilyd@gmail.com`) є адреса електронної пошти `thigg@gmail.com`

і службова адреса електронної пошти thomas.higginson@gmail.com, то постачальник контактів зберігає два рядки адреси електронної пошти та пов'язує їх із цим необробленим контактом.

Зверніть увагу, що в цій таблиці зберігаються дані різних типів. Рядки з відомостями про назву, номер телефону, адресу електронної пошти, поштову адресу, фото та веб-сайт, що відображаються, зберігаються в таблиці ContactsContract.Data. Для спрощення управління цими даними в таблиці ContactsContract.Data передбачено стовпці з описовими назвами, а також інші стовпці з універсальними назвами. Вміст у стовпці з описовою назвою має те ж значення, незалежно від типу даних в рядку, тоді як вміст стовпчика з універсальною назвою може мати різне значення, залежно від типу даних.

Описові назви стовпців

Ось деякі приклади стовпців з описовими назвами:

RAW_CONTACT_ID. Значення у стовпці _ID необробленого контакту для цих даних;

MIMETYPE. Тип даних, що зберігаються в цьому рядку у вигляді типу MIME, що налаштовуються. Постачальник контактів використовує типи MIME, задані в підкласах класу ContactsContract.CommonDataKinds. Ці типи MIME є відкритими, і їх може використовувати будь-який додаток або адаптер синхронізації, що підтримує роботу з постачальником контактів;

IS_PRIMARY. Якщо цей тип даних для необробленого контакту зустрічається кілька разів, то стовпець IS_PRIMARY позначає прапорцем рядки даних, у яких містяться основні дані для цього типу.

Наприклад, якщо користувач натиснув і утримує номер телефону контакту та вибрав параметр *Використовувати за умовчуванням*, то у стовпці ContactsContract.Data із цим номером телефону у відповідному стовпці IS_PRIMARY задається значення, відмінне від нуля.

Універсальні назви стовпців

Існує 15 загальнодоступних стовпців з універсальними назвами (DATA1 – DATA15) і чотири додаткових стовпчика (SYNC1 – SYNC4), які використовуються тільки адаптерами синхронізації. Константи стовпців з універсальними назвами застосовують завжди, незалежно від типу даних, що містяться у стовпці.

Стовпець DATA1 є індексованим. Постачальник контактів завжди використовує цей стовпець для даних, які, як він очікує, будуть цільовими в запитах. Наприклад, у рядку з адресами електронної пошти в цьому стовпці вказано фактичну адресу електронної пошти.

Зазвичай, стовпець DATA15 зарезервовано для даних таких великих двійкових об'єктів (BLOB), як мініатюри фотографій.

Назви стовпців за типами рядків

Для спрощення роботи за допомогою стовпців певного типу рядків у постачальнику контактів також передбачено константи для назв стовпців за типами рядків, які визначені в підкласах класу ContactsContract.CommonDataKinds. Ці константи просто надають одній і тієї ж назві стовпця різні константи, що дозволяє отримувати доступ до даних у рядку певного типу.

Наприклад, клас ContactsContract.CommonDataKinds.Email визначає константи назви стовпця для рядка ContactsContract.Data, у якому є тип MIME Email.CONTENT_ITEM_TYPE. У цьому класі міститься константа ADDRESS для стовпця адреси електронної пошти. Фактичне значення ADDRESS – data1, яке збігається з універсальною назвою стовпця.

Увага! Не слід додавати свої дані, що налаштовуються, у таблицю ContactsContract.Data за допомогою рядка, у якому є один із попередньо визначених постачальником типів MIME. В іншому випадку можна втратити дані або викликати несправності в роботі постачальника. Наприклад, не слід додавати рядок із типом MIME Email.CONTENT_ITEM_TYPE, у якому у стовпці DATA1 замість адреси електронної пошти міститься назва користувача. Якщо вказати в рядку власний тип MIME, що налаштовується, можна вільно вказувати власні назви стовпців за типами рядків і використовувати їх так, як забажаєте.

На рис. 4.39 показано, як стовпці з описовими назвами та стовпці даних відображаються в рядку ContactsContract.Data, а також як назви стовпців за типом рядків "накладаються" на універсальні назви стовпців.

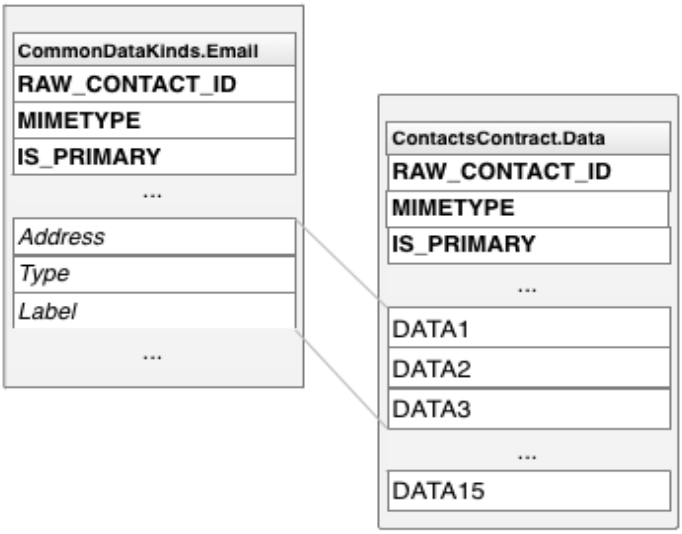

Рис. 4.39. **Імена стовпців за типами рядків та універсальні назви стовпців**

Класи назв стовпців за типами рядків

У табл. 4.5 перелічено класи назв стовпців за типами рядків, які частіше за все використовують.

Таблиця 4.5

Класи назв стовпців за типами рядків

`ContactsContract.CommonDataKinds`

Класи зіставлення	Типи даних	Примітки
`.StructuredName`	Дані про назву необробленого контакту, пов'язаного із цим рядком даних	У необробленого контакту є тільки один рядок такого типу
`.Photo`	Основна фотографія необробленого контакту, пов'язаного із цим рядком даних	У необробленого контакту є тільки один рядок такого типу
`.Email`	Адреса електронної пошти необробленого контакту, пов'язаного із цим рядком даних	У необробленого контакту може бути декілька адрес електронної пошти
`.StructuredPostal`	Поштова адреса необробленого контакту, пов'язаного із цим рядком даних	У необробленого контакту може бути декілька поштових адрес
`.GroupMembership`	Ідентифікатор, за допомогою якого необроблений контакт пов'язано з однією із груп у постачальника контактів	Групи є необов'язковим компонентом для типу акаунту та назви акаунту

Контакти

Постачальник контактів об'єднує рядки з необробленими контактами для всіх типів акаунтів та назв акаунтів для створення **контакту**. Це дозволяє спростити відображення та зміну всіх даних, зібраних щодо користувача. Постачальник контактів управляє процесом створення рядків контактів та агрегуванням необроблених контактів, у яких є рядок контакту. Ні додаткам, ні адаптерам синхронізації не дозволено додавати контакти, а деякі стовпці в рядку контакту доступні тільки для читання.

Примітка. Якщо спробувати додати контакт у постачальник контактів за допомогою методу `insert()`, буде видано виняток `UnsupportedOperation Exception`. Якщо спробувати оновити стовпець, доступний тільки для читання, цю дію буде проігноровано.

Під час додавання нового необробленого контакту, який не відповідає жодному з наявних контактів, постачальник контактів створює новий контакт. Постачальник чинить аналогічно в разі, якщо дані в рядку наявного необробленого контакту змінюються таким чином, що вони більше не відповідають контакту, із яким вони раніше були пов'язані. Під час створення додатком або адаптером синхронізації нового контакту, який відповідає наявному контакту, новий контакт об'єднується з наявним контактом.

Постачальник контактів пов'язує рядок контакту з рядками необробленого контакту за допомогою стовпчика `_ID` рядка контакту в таблиці `Contacts`. Стовпець `CONTACT_ID` у таблиці необроблених контактів `Contacts Contract.RawContacts` містить значення `_ID` для рядка контактів, пов'язаного з кожним рядком необроблених контактів.

У таблиці `ContactsContract.Contacts` також є стовпець `android.provider. ContactsContract.ContactsColumns#LOOKUP_KEY`, який є "постійним посиланням" на рядок контакту. Оскільки постачальник контактів автоматично зберігає контакти, у відповідь на агрегування або синхронізацію, він може змінити значення `android.provider.BaseColumns#_ID` рядка контакту. Навіть якщо це станеться, URI контенту `CONTENT_LOOKUP_URI`, об'єднаний з `android.provider. ContactsContract.ContactsColumns#LOOKUP_KEY` контакту, буде як і раніше вказувати на рядок контакту, тому можна сміливо використовувати `android.provider.ContactsContract.ContactsColumns#LOOKUP_KEY` для збереження посилань на "обрані" контакти та ін. Стовпець має власний формат, який не пов'язано з форматом стовпчика `android.provider.BaseColumns#_ID`.

На рис. 4.40 показано взаємозв'язок цих трьох основних таблиць одна з одною.

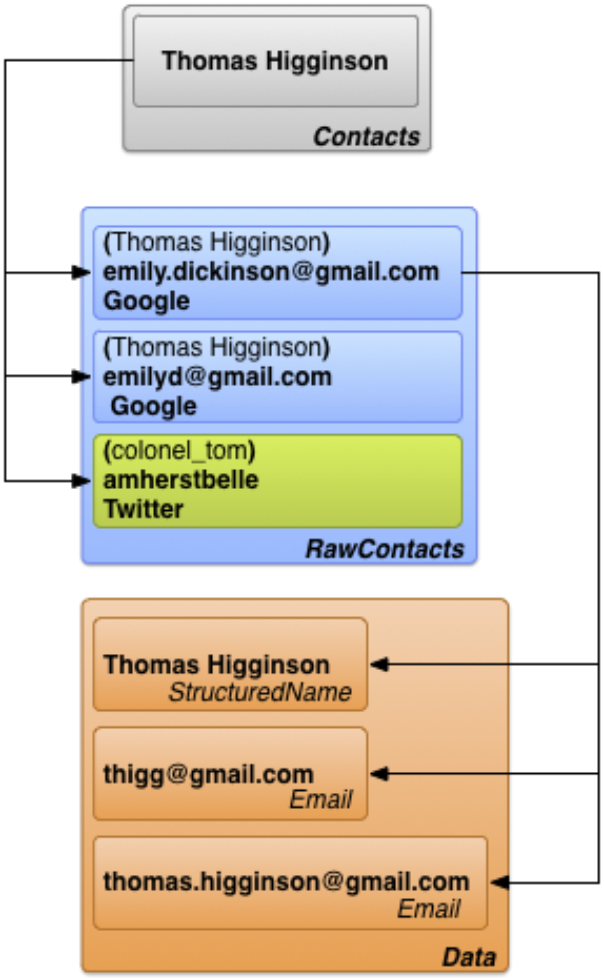

Рис. 4.40. **Взаємозв'язок між таблицями контактів, необроблених контактів і відомостей**

4.4.4. Дані, отримані від адаптера синхронізації

Користувач уводить дані контактів прямо на пристрої, проте дані також надходять у постачальник контактів із веб-служб за допомогою **адаптерів синхронізації**, що дозволяє автоматизувати обмін даними між пристроєм і службами в Інтернеті. Адаптери синхронізації виконуються у фоновому режимі під управлінням системи, і для управління даними вони викликають методи `ContentResolver`.

В Android веб-службу, із якою працює адаптер синхронізації, визначено за типом акаунту. Кожен адаптер синхронізації працює з одним типом акаунту, проте він може підтримувати кілька назв акаунтів такого типу. Зазначені далі визначення дозволяють точніше охарактеризувати зв'язок між типами й назвами акаунтів і адаптерами синхронізації і службами.

Тип акаунту. Визначає службу, у якій користувач зберігає дані. У більшості випадків для роботи зі службою користувачеві необхідно пройти перевірку справжності. Наприклад, *Google Контакти* становить тип акаунтів, що позначено кодом google.com. Це значення відповідає типу акаунту, використовуваного `AccountManager`.

Назва акаунту. Визначає конкретний акаунт або назву для входу для типу акаунту. Акаунти "*Контакти Google*" – це те саме, що й акаунти Google, іменем яких використовано адресу електронної пошти. В інших службах може використовуватися назва користувача, що складається з одного слова, або числовий ідентифікатор.

Типи акаунтів не обов'язково мають бути унікальними. Користувач може створити кілька облікових записів *Google Контакти* та завантажити дані з них у постачальник контактів; це може статися в разі, якщо у користувача є один набір персональних контактів для особистого облікового запису, а інший набір – для службового акаунту. Назви акаунтів, зазвичай, унікальні. Разом вони формують певний потік даних між постачальником контактів і зовнішніми службами.

Якщо необхідно передати дані зі служби в постачальник контактів, необхідно створити власний адаптер синхронізації.

На рис. 4.41. показано, яку роль виконує постачальник контактів у потоці передачі даних про користувачів. В області, зазначеній на рисунку як *sync adapters*, кожен адаптер промарковано, відповідно до підтримуваних ним типів акаунтів.

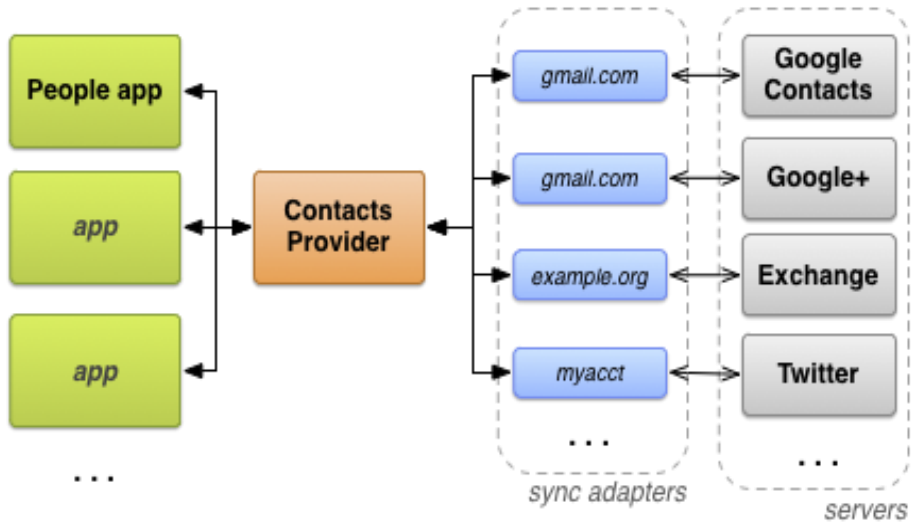

Рис. 4.41. **Потік передачі даних через постачальника контактів**

4.4.5. Необхідні дозволи

Додатки, яким потрібен доступ до постачальника контактів, мають запросити такі дозволи:

1. Доступ на читання однієї або декількох таблиць `READ_CONTACTS`, указаний у файлі `AndroidManifest.xml` з елементом `<uses-permission>` у вигляді `<uses-permission android:name="android.permission.READ_CONTACTS">`.

2. Доступ на запис в одну або кілька таблиць `WRITE_CONTACTS`, указаний у файлі `AndroidManifest.xml` з елементом `<uses-permission>` у вигляді `<uses-permission android:name="android.permission.WRITE_CONTACTS">`.

Ці дозволи не поширюються на роботу з даними профілю користувача.

Слід пам'ятати, що дані про контакти користувача є особистими та конфіденційними. Користувачі піклуються про свою конфіденційність, тому не хочуть, щоб програми збирали дані про них самих або їхніх контактах. Якщо користувачеві не до кінця ясно, для чого потрібен дозвіл на доступ до даних контактів, вони можуть надати додатку низький рейтинг або взагалі відмовляться його встановлювати.

4.4.6. Профіль користувача

У таблиці `ContactsContract.Contacts` є всього один рядок, що містить дані профілю для пристрою користувача. Ці дані описують user пристрою, а не один із контактів користувача. Рядок контактів профілю пов'язано з рядком необроблених контактів у кожній із систем, у якій використовують профіль. У кожному рядку необробленого контакту профілю може міститися декілька рядків даних. Константи для доступу до профілю користувача наведено у класі `ContactsContract.Profile`.

Для доступу до профілю користувача потрібні особливі дозволи. Крім дозволів `READ_CONTACTS` та `WRITE_CONTACTS`, які потрібні для читання й запису, щоб отримати доступ до профілю користувача необхідні дозволи `android.Manifest.permission#READ_PROFILE` та `android.Manifest.permission#WRITE_PROFILE` на читання й запис, відповідно.

Завжди слід пам'ятати, що профіль користувача становить конфіденційну інформацію. Дозвіл `android.Manifest.permission#READ_PROFILE` надає вам доступ до особистої інформації на пристрої користувача. В описі свого додатка обов'язково слід вказати, для чого потрібен доступ до профілю користувача.

Щоб отримати рядок контакту із профілем користувача, слід викликати метод `ContentResolver.query()`. Потрібно задати для URI контенту значення `CONTENT_URI` та не вказувати ніяких критеріїв вибірки. Цей URI контенту також можна використовувати як основний URI для отримання необроблених контактів або деталей свого профілю. Далі наведено фрагмент коду для отримання даних профілю (рис. 4.42):

```
// Установіть стовпці для вилучення для профілю користувача
mProjection = new String[]
    {
        Profile._ID,
        Profile.DISPLAY_NAME_PRIMARY,
        Profile.LOOKUP_KEY,
        Profile.PHOTO_THUMBNAIL_URI
    };

// Витягує профіль із контактів постачальника
mProfileCursor =
        getContentResolver().query(
                Profile.CONTENT_URI,
                mProjection ,
                null,
                null,
                null);
```

Рис. 4.42. **Отримання даних профілю**

Примітка. Якщо під час витягання кількох рядків контактів необхідно визначити, який із них є профілем користувача, слід звернутися до колонки `IS_USER_PROFILE` цього рядка. Якщо в цьому стовпці вказано значення 1, то в цьому рядку знаходиться профіль користувача.

4.4.7. Метадані постачальника контактів

Постачальник контактів управляє даними, які використовують для відстеження стану контактної інформації в репозиторії. Ці метадані про репозиторії зберігаються в різних місцях, включаючи рядки необроблених контактів, дані та рядки таблиці контактів, таблиці `ContactsContract.Settings` і `ContactsContract.SyncState`. У табл. 4.6 далі вказано значення кожного із цих фрагментів метаданих.

Таблиця 4.6

Метадані в постачальнику контактів `ContactsContract`

Таблиці	Стовпці	Значення	Описи
1	2	3	4
`RawContacts`	`DIRTY`	"0" – із моменту останньої синхронізації зміни не вносилися	Слугує для позначення необроблених контактів, які було змінено на пристрої та які необхідно знову синхронізувати із сервером. Значення встановлюється автоматично постачальником контактів під час оновлення рядків додатками Android. Адаптери синхронізації, які вносять зміни в необроблені контакти або таблиці даних, мають завжди додавати до використовуваного ними URI контенту рядок `CALLER_IS_SYNCADAPTER`. Це дозволяє запобігти позначенню таких рядків постачальником як "брудних". В іншому випадку зміни, внесені адаптером синхронізації, будуть розглядати як локальні зміни, які підлягають відправленню на сервер, навіть якщо джерелом таких змін був сам сервер
		"1" – із моменту останньої синхронізації було внесено зміни, які необхідно відобразити й на сервері	
`RawContacts`	`VERSION`	Номер версії цього рядка	Постачальник контактів автоматично збільшує це значення під час кожної зміни рядку або пов'язаних із ним даних
		Номер версії цього рядка	Постачальник контактів автоматично збільшує це значення під час кожної зміни в рядку даних
`RawContacts`	`SOURCE_ID`	Рядкове значення, яке є унікальним ідентифікатором цього необробленого контакту в акаунті, у якому його було створено	Коли адаптер синхронізації створює новий необроблений контакт, в цьому стовпці слід указати унікальний ідентифікатор цього необробленого контакту на сервері. Коли ж додаток Android створює новий необроблений контакт, то у додатку слід залишити цей стовпець порожнім. Це слугує

Продовження табл. 4.6

1	2	3	4
			сигналом для адаптера синхронізації для створення нового необробленого контакту на сервері й отримання значення `SOURCE_ID`. Зокрема, ідентифікатор джерела має бути **унікальним** для кожного типу акаунту та незмінним під час синхронізації. **Унікальний:** у кожного необробленого контакту для акаунту має бути свій власний ідентифікатор джерела. Якщо не застосувати цю вимогу, обов'язково виникнуть проблеми з додатком *Контакти*. Слід звернути увагу, що два необроблених контакти одного й того ж *типу* акаунту можуть мати однаковий ідентифікатор джерела. Наприклад, необроблений контакт Thomas Higginson для акаунту `emily.dickinson@gmail.com` може мати такий же ідентифікатор джерела, що і контакт Thomas Higginson для акаунту `emilyd@gmail.com`. **Стабільний:** ідентифікатори джерел становлять незмінну частину даних онлайн-служби для необробленого контакту. Наприклад, якщо користувач виконує повторну синхронізацію після очищення сховища контактів в налаштуваннях програми, ідентифікатори джерел відновлених необроблених контактів мають бути такими ж, як і раніше. Якщо не застосувати цю вимогу, перестануть працювати ярлики
`Groups`	`GROUP_VISIBLE`	"0" – контакти, наведені в цій групі, не мають відображатися в інтерфейсах користувача додатків Android	Цей стовпець призначено для відомостей про сумісність з серверами, що дозволяють користувачам приховувати контакти в певних групах

Закінчення табл. 4.6

1	2	3	4
		"1" – контакти, наведені в цій групі, можуть відображатися в інтерфейсах користувача додатків	
Settings	UNGROUPED_VISIBLE	"0" – для цього облікового запису й акаунтів цього типу контактів, які не належать до групи, не відображаються в інтерфейсах користувача додатків Android	За замовчуванням контакти приховані, якщо жоден із їх необроблених контактів не належить групі (належність необробленого контакту до групи вказано в одному або декількох рядках ContactsContract.CommonDataKinds.GroupMembership у таблиці ContactsCont ract.Data). Установивши цей прапорець у рядку таблиці Conta ctsContract.Settings для типу акаунту та назви акаунту, можна примусово зробити видимими контакти, які не належать будь-якій групі. Один із варіантів використання цього прапорця – відображення контактів із серверів, на яких не використовують групи
		"1" – для цього облікового запису і акаунтів цього типу контакти, які не належать до групи, відображаються в інтерфейсах користувача додатків Android	–
SyncState	(усі)	Цю таблицю використовують для зберігання метаданих для адаптера синхронізації	За допомогою цієї таблиці можна на постійній основі зберігати на пристрої відомості про стан синхронізації та інші пов'язані із синхронізацією дані

4.4.8. Доступ до постачальника контактів

У цьому підрозділі розглядають інструкції щодо отримання доступу до даних із постачальника контактів, зокрема, такі, як:

запити об'єктів;

пакетна зміна;

отримання та зміна даних за допомогою намірів;

цілісність даних.

Запит об'єктів

Таблиці постачальника контактів мають ієрархічну структуру, тому часто корисно витягати рядок і всі пов'язані з ним його дочірні рядки. Наприклад, для відображення всієї інформації про користувача, потрібно отримати всі рядки `ContactsContract.RawContacts` для одного рядка `ContactsContract.Contacts` або всі рядки `ContactsContract.CommonDataKinds.Email` для одного рядка `ContactsContract.RawContacts`. Щоб спростити цей процес, у постачальника контактів є **об'єкти**, які є з'єднувачами бази даних між таблицями.

Об'єкт становить подобу таблиці, що складається з обраних стовпців батьківської таблиці та її дочірньої таблиці. Під час запиту об'єкта слід надати проекцію та критерії пошуку на основі доступних в об'єкті стовпців. У результаті отримано об'єкт `Cursor`, у якому міститься один рядок для кожного витягнутого рядка дочірньої таблиці. Наприклад, якщо запросити об'єкт `ContactsContract.Contacts.Entity` для назви контакту та всі рядки `ContactsContract.CommonDataKinds.Email` для всіх необроблених контактів, що відповідають цієї назві, отримують об'єкт `Cursor`, що містить по одному рядку для кожного рядка `ContactsContract.CommonDataKinds.Email`.

Використання об'єктів спрощує запити. За допомогою об'єкта можна витягти відразу всі дані для контакту або необробленого контакту, замість того, щоб спочатку робити запит до батьківської таблиці для отримання ідентифікатора й подальшого запиту до дочірньої таблиці з використанням отриманого ідентифікатора. Крім того, постачальник контактів обробляє запит об'єкта за одну транзакцію, що гарантує внутрішню погодженість отриманих даних.

Примітка. Об'єкт, зазвичай, містить не всі стовпці батьківської та дочірньої таблиць. Якщо спробувати змінити назву стовпця, який відсутній у списку констант назви стовпця для об'єкта, буде отримано `Exception`.

Далі наведено приклад коду для отримання всіх рядків необробленого контакту для контакту. Це фрагмент більшого додатка, у якому є дві операції: *основна* та *для отримання відомостей*. Основна операція слугує для отримання списку рядків контактів; під час вибору користувачем контакту операція відправляє його ідентифікатор в операцію для отримання відомостей. Операція для отримання відомостей використовує об'єкт `ContactsContract.Contacts.Entity` для відображення всіх рядків даних, отриманих від усіх необроблених контактів, пов'язаних з обраним контактом.

Ось фрагмент коду операції *для отримання відомостей* (рис. 4.43):

```
...
  /* Додає шлях сутності до URI. У разі постачальника контактів, очікується URI змісту://com.google.contacts/#/entity (# це значення ідентифікатора). */
  mContactUri = Uri.withAppendedPath(
            mContactUri,
            ContactsContract.Contacts.Entity.CONTENT_DIRECTORY);

  // Ініціалізовує завантажувач, ідентифікований за LOADER_ID.
  getLoaderManager().initLoader(
         LOADER_ID,  // Ідентифікатор завантажувача для ініціалізації
         null,    // Аргументи для завантажувача (у цьому випадку, немає)
         this);   // контекст діяльності


  // Створює новий адаптер курсора, щоб прикріпити до перегляду списку
  mCursorAdapter = new SimpleCursorAdapter(
    this,                        // контекст діяльності
    R.layout.detail_list_item, // вид елемента, що містить деталі віджетів
    mCursor,           // курсор заднього плану
    mFromColumns,     // стовпці в курсорі, які надають дані
    mToViews,         // види в пункті уявлення , що відображають дані
    0);               // прапорці

    // Установлює адаптер ListView.
    mRawContactList.setAdapter(mCursorAdapter);
...
```

Рис. 4.43. **Операції для отримання відомостей**

```
@Override
public Loader<Cursor> onCreateLoader(int id, Bundle args) {

  /* Установлює стовпці для вилучення.
  * RAW_CONTACT_ID включено для ідентифікації необробленого контакту,
  * пов'язаного з рядком даних.
  * DATA1 містить перший стовпець у рядку даних (як правило,
  * найважливіший із них).
  * MIMETYPE вказує тип даних у рядку даних.
  */
  String[] projection =
        {
            ContactsContract.Contacts.Entity.RAW_CONTACT_ID,
            ContactsContract.Contacts.Entity.DATA1,
            ContactsContract.Contacts.Entity.MIMETYPE
        };

  /* Проведіть сортування витягнутого курсора необробленого ID контакта,
щоб зберегти всі рядки даних для одного контакту, що зіставляються один
з одним. */
  String sortOrder = ContactsContract.Contacts.Entity.RAW_CONTACT_ID +
            " ASC";

  /* Виконує новий CursorLoader. Аргументи аналогічні
ContentResolver.query(), крім контексту аргументу, який постачає місце
проведення ContentResolver */
  return new CursorLoader(
        getApplicationContext(),  // контекст діяльності
        mContactUri,         // Зміст об'єкта URI для одного контакту
        projection,          // Колонки для вилучення
        null,     // Отримати всі вихідні контакти та їхні ряди даних
        null,     //
        sortOrder);       // Сортування за ID необробленого контакту.
}
```

Закінчення рис. 4.43

Після завершення завантаження `LoaderManager` виконує зворотний виклик методу `onLoadFinished()`. Одним із вхідних аргументів для цього методу є `Cursor` із результатом запиту. У своєму додатку можна отримати дані із цього об'єкта `Cursor` для його відображення або подальшої роботи з ним.

Пакетне перетворення. При кожній нагоді дані в постачальнику контактів слід уставляти, оновлювати та видаляти в *пакетному режимі* шляхом створення `ArrayList` з об'єктів `ContentProviderOperation` і виклику методу `applyBatch()`. Оскільки постачальник контактів виконує всі операції в методі `applyBatch()` за одну транзакцію ще раз, то зміни завжди будуть знаходитися в межах сховища контактів і будуть погодженими. Пакетне перетворення також спрощує вставку необробленого контакту одночасно з його детальними даними.

Примітка. Щоб змінити *окремий* необроблений контакт, рекомендовано відправити намір у додаток для роботи з контактами на пристрої, замість оброблення зміни в додатку.

Межі. Пакетне перетворення великої кількості операцій може заблокувати виконання інших процесів, що може негативно позначитися на роботі користувача з додатком. Щоб упорядкувати всі необхідні зміни в межах якомога меншої кількості окремих списків і до того ж запобігти блокуванню роботи системи, слід задати **межі** для однієї або декількох операцій. Межа є об'єктом `ContentProviderOperation`, значенням параметра `isYieldAllowed()` якого задано `true`. Під час досягнення постачальником контактів межі він призупиняє свою роботу, щоб могли виконувати інші процеси, і закриває поточну транзакцію. Коли постачальник запускається знову, він виконує наступну операцію в `ArrayList` і запускає нову транзакцію.

Використання меж дозволяє за один виклик методу `applyBatch()` виконувати більше від однієї транзакції. Із цієї причини слід задати межі для останньої операції з набором пов'язаних рядків. Наприклад, необхідно задати межі для останньої операції в наборі, яка додає рядки необробленого контакту та пов'язані з ним рядки даних, або для останньої операції з набором рядків, пов'язаних з одним контактом.

Межі також є одиницями атомарних операцій. Усі операції доступу у проміжку між двома межами завершуються або успіхом, або перебоями в межах однієї одиниці. Якщо будь-яку з меж не задано, найменшою атомарною операцією вважають увесь пакет операцій. Використання меж дозволяє запобігти зниженню продуктивності системи й одночасно забезпечити виконання набору операцій на атомарному рівні.

Зміна зворотних посилань. Під час уставлення нового рядка необробленого контакту та пов'язаних із ним рядів даних у вигляді набору

об'єктів ContentProviderOperation доводиться пов'язувати рядки даних із рядком необробленого контакту шляхом установлення значення android.provider.BaseColumns#_ID необробленого контакту у вигляді значення RAW_CONTACT_ID. Однак це значення недоступне, поки створюють ContentProviderOperation для рядка даних, оскільки ще не застосували ContentProviderOperation для рядка необробленого контакту. Щоб обійти це обмеження, у класі ContentProviderOperation.Builder передбачено метод withValueBackReference(). За допомогою цього методу можна встановлювати або змінювати стовпець із результатом попередньої операції.

У методі withValueBackReference() передбачено два аргументи:

key – ключ для пари "ключ-значення". Значенням цього аргументу має бути назвою стовпця в таблиці, яку змінюють;

previousResult – індекс значення, що починається з "0", у масиві об'єктів ContentProviderResult із методу applyBatch(). По мірі виконання пакетних операцій результат кожної операції зберігається у проміжному масиві результатів. Значенням previousResult є індекс одного із цих результатів, який витягується та зберігається зі значенням key. Завдяки цьому можна вставити новий запис необробленого контакту та отримати назад його значення android.provider.BaseColumns#_ID, а потім створити *зворотне посилання* на значення під час додавання рядка Contacts Contract.Data.

Цілком увесь результат створюється за першого виклику методу applyBatch(). Розмір результату дорівнює розміру ArrayList наданих об'єктів ContentProviderOperation. Однак усім елементам у масиві результатів надають значення null, і за спроби скористатися зворотним посиланням на результат операції, що ще не виконалася, метод withValueBack Reference() видає Exception.

Далі наведено фрагменти коду для вставлення нового необробленого контакту та його даних у пакетному режимі. Вони містять код, який задає межу та використовує зворотне посилання. Ці фрагменти становлять розширену версію методу createContactEntry(), який входить до класу ContactAdder у прикладі застосування Contact Manager.

Перший фрагмент коду слугує для отримання даних контакту із інтерфейсу користувача. На цьому етапі користувач уже вибрав акаунт, для якого необхідно додати новий необроблений контакт (рис. 4.44).

```
// Створює контакт із поточних значень інтерфейсу користувача
з використанням поточного обраного рахунка.
protected void createContactEntry() {
    /*
     * Отримує значення із інтерфейсу користувача
     */
    String name = mContactNameEditText.getText().toString();
    String phone = mContactPhoneEditText.getText().toString();
    String email = mContactEmailEditText.getText().toString();

    int phoneType = mContactPhoneTypes.get(
            mContactPhoneTypeSpinner.getSelectedItemPosition());

    int emailType = mContactEmailTypes.get(
            mContactEmailTypeSpinner.getSelectedItemPosition());
```

Рис. 4.44. **Отримання даних контакту**

У наступному фрагменті коду створюється операція для вставлення рядка необробленого контакту в таблицю `ContactsContract.RawContacts` (рис. 4.45):

```
/* Підготовлює пакетне оброблення для вставлення нового необробленого
контакту та його дані. Навіть якщо Постачальник Контактів не має ніяких
даних для цієї людини, ви не можете додати контакт, тільки необроблений
контакт. Постачальник Контактів потім додасть контакт автоматично. */

// Створює новий масив об'єктів ContentProviderOperation.
ArrayList<ContentProviderOperation> ops =
            new ArrayList<ContentProviderOperation>();

/* Створює новий необроблений контакт із його типом облікового запису
(тип сервера) та назву облікового запису (рахунок користувача). Пам'я-
тайте, що псевдонім не зберігається в цьому рядку, а в рядку даних
StructuredName. Інші дані не потрібні. */
ContentProviderOperation.Builder op = ContentProviderOperation
   .newInsert(ContactsContract.RawContacts.CONTENT_URI)
   .withValue(ContactsContract.RawContacts.ACCOUNT_TYPE,
              mSelectedAccount.getType())
```

Рис. 4.45. **Створення операції для вставлення рядка необробленого контакту в таблицю `ContactsContract.RawContacts`**

```
    .withValue(ContactsContract.RawContacts.ACCOUNT_NAME,
               mSelectedAccount.getName());

// Будує операцію та додає її до масиву операцій
ops.add(op.build());
```

Закінчення рис. 4.45

Потім код створює рядки даних для рядків імені, телефону та адреси електронної пошти, що відображуються.

Кожен об'єкт компонента операції використовує метод `withValue BackReference()` для отримання `RAW_CONTACT_ID`. Посилання повертається назад до об'єкта `ContentProviderResult` із першої операції, у результаті чого додається рядок необробленого контакту та повертається його нове значення `android.provider.BaseColumns#_ID`. Після цього кожен рядок даних автоматично пов'язується за своїм `RAW_CONTACT_ID` із новим рядком `ContactsContract.RawContacts`, до якого він належить.

Об'єкт `ContentProviderOperation.Builder`, який додає рядок адреси електронної пошти, позначається прапорцем за допомогою методу `withYieldAllowed()`, який задає межі (рис. 4.46):

```
/* Створює назву для нового необробленого контакту як ряд даних
StructuredName. */
op = ContentProviderOperation
    .newInsert(ContactsContract.Data.CONTENT_URI)

/* з ValueBackReference встановлює значення першого аргументу до значення
ContentProviderResult, що індексується другим аргументом. У цьому конк-
ретному виклику, ID колонки необробленого контакту StructuredName рядок
даних встановлюється на значення результату, що повертається першою опе-
рацією, саме яка фактично додає необроблений рядок контакту. */
    .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, 0)

    // Установлює тип даних MIME до StructuredName
    .withValue(ContactsContract.Data.MIMETYPE,
      ContactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE)

    // Задає коротку назву рядка даних для назви в інтерфейсі.
    .withValue(
          ContactsContract.CommonDataKinds.StructuredName.DISPLAY_NAME,
```

Рис. 4.46. **Реалізація об'єкту `ContentProviderOperation.Builder`**

```
            name);

// Будує операцію та додає її в масив операцій
ops.add(op.build());

// Уставляє вказаний телефонний номер і тип як рядок даних телефону
op =ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI)
    /*
    * Установлює значення вихідного стовпця ідентифікатора контакту до
    * нового необробленого ID-контакту, повернутого першою операцією в
    * пакеті.
    */
    .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, 0)

    // Задає рядок даних MIME-типу для телефону
    .withValue(ContactsContract.Data.MIMETYPE,
               ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE)

    // Установлює номер і тип телефону
    .withValue(ContactsContract.CommonDataKinds.Phone.NUMBER, phone)
    .withValue(ContactsContract.CommonDataKinds.Phone.TYPE, phoneType);

// Будує операцію та додає її в масив операцій
ops.add(op.build());

/* Уставляє вказану адресу електронної пошти та тип як рядок даних теле-
фону */
op = ContentProviderOperation.newInsert(
                                    ContactsContract.Data.CONTENT_URI)
    /*
    * Установлює значення вихідного стовпця ідентифікатора контакту до
    * нового необробленого ID-контакту, повернутого першою операцією в
    * пакеті.
    */
    .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, 0)

    // Задає рядок даних MIME-типу для електронної адреси
    .withValue(ContactsContract.Data.MIMETYPE,
               ContactsContract.CommonDataKinds.Email.CONTENT_ITEM_TYPE)

    // Установлює електронну адресу та тип
```

Продовження рис. 4.46

```
    .withValue(ContactsContract.CommonDataKinds.Email.ADDRESS, email)
    .withValue(ContactsContract.CommonDataKinds.Email.TYPE, emailType);

/* Демонструється вихідна точка. Після закінчення цього вставлення, пар-
тія пакетних операцій дасть пріоритет для інших потоків. Використовуйте
після кожного набору операцій, які впливають на один контакт, щоб уник-
нути зниження продуктивності. */
op.withYieldAllowed(true);

// Будує операцію та додає її в масив операцій
ops.add(op.build());
```

Закінчення рис. 4.46.

В останньому фрагменті коду наведено виклик методу `applyBatch()` для вставлення нового необробленого контакту та його рядків даних (рис. 4.47):

```
// Попросіть постачальника контактів створити новий контакт
Log.d(TAG,"Selected account: " + mSelectedAccount.getName() + " (" +
            mSelectedAccount.getType() + ")");
Log.d(TAG,"Creating contact: " + name);

/* Застосовує масив об'єктів ContentProviderOperation у пакетному режи-
мі. Результати відкидаються. */
try {
  getContentResolver().applyBatch(ContactsContract.AUTHORITY, ops);
} catch (Exception e) {
  // Показує застреження
  Context ctx = getApplicationContext();
  CharSequence txt = getString(R.string.contactCreationFailure);
  int duration = Toast.LENGTH_SHORT;
  Toast toast = Toast.makeText(ctx, txt, duration);
  toast.show();

  // Уведення застереження
  Log.e(TAG, "Exception encountered while inserting contact: " + e);
}
}
```

Рис. 4.47. **Вставлення нового необробленого контакту та його рядків даних**

За допомогою пакетних операцій можна також реалізувати **оптимістичне управління паралелізмом**. Це метод застосування транзакцій зміни без необхідності блокувати базовий репозиторій. Щоб скористатися цим методом, необхідно застосувати транзакцію і перевірити наявність інших змін, які могли бути внесені в цей же час. Якщо виявиться, що є непогоджена зміна, транзакцію можна відкотити та виконати повторно.

Оптимістичне управління паралелізмом підходить для мобільних пристроїв, де із системою одночасно взаємодіє тільки один користувач, а одночасний доступ до сховища даних декількох користувачів – досить рідкісне явище. Оскільки не застосовується блокування, економиться дорогоцінний час, який потрібен на встановлення блокувань або очікування того, коли інші транзакції скасують свої блокування.

Далі наведено порядок використання оптимістичного управління паралелізмом під час оновлення одного рядка `ContactsContract.RawContacts`.

1. Вийняти стовпець `VERSION` необробленого контакту, а також інші дані, які необхідно витягти.

2. Створити об'єкт `ContentProviderOperation.Builder`, відповідний для застосування обмеження, за допомогою методу `newAssertQuery(Uri)`. Для URI контенту слід використовувати `RawContacts.CONTENT_URI`, додавши до нього `android.provider.BaseColumns#_ID` необробленого контакту.

3. Для об'єкта `ContentProviderOperation.Builder` викликати метод `withValue()`, щоб порівняти стовпець `VERSION` із номером версії, яку тільки що отримали.

4. Для того ж об'єкта `ContentProviderOperation.Builder` викликати метод `withExpectedCount()`, щоб переконатися в тому, що перевірне твердження перевіряє тільки один рядок.

5. Викликати метод `build()`, щоб створити об'єкт `ContentProvider Opera tion`, потім додати цей об'єкт першим в `ArrayList`, який передати в метод `applyBatch()`.

6. Застосувати пакетну транзакцію.

Якщо в проміжку між зчитуванням рядка та спробою його зміни рядок необробленого контакту було оновлено іншою операцією, *assert* `ContentProviderOperation` завершиться перебоями та у виконанні всього пакета операцій буде відмовлено. Можна вибрати повторне виконання пакета або виконати іншу дію.

У прикладі коду далі продемонстровано, як створити assert `ContentProviderOperation` після запиту одного необробленого контакту за допомогою `CursorLoader` (рис. 4.48):

```
/* Додаток використовує CursorLoader для того, щоб запросити таблицю
необроблених контактів. Система викликає цей метод коли закінчиться за-
вантаження. */
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
  // Отримує _ID і VERSION значення необробленого контакту
  mRawContactID = cursor.getLong(cursor.getColumnIndex(BaseColumns._ID));
  mVersion = cursor.getInt(cursor.getColumnIndex(SyncColumns.VERSION));
}
...

// Установлює Uri для операції Assert
Uri rawContactUri = ContentUris.withAppendedId(RawContacts.CONTENT_URI,
                                               mRawContactID);

// Створює конструктор для операції Assert
ContentProviderOperation.Builder assertOp =
                ContentProviderOperation.netAssertQuery(rawContactUri);

/* Додає затвердження до операції Assert: перевіряє версію і кількість
рядків, що тестуються */
assertOp.withValue(SyncColumns.VERSION, mVersion);
assertOp.withExpectedCount(1);

// Створює ArrayList для зберігання об'єктів ContentProviderOperation
ArrayList ops = new ArrayList<ContentProviderOperationg>;

ops.add(assertOp.build());

// Ви можете додати решту ваших пакетних операцій до «OPS» тут

...
```

Рис. 4.48. **Створення assert `ContentProviderOperation` після запиту одного необробленого контакту за допомогою `CursorLoader`**

```
// Використовує групу. Якщо ствердження провалюється, виникає виключення
try {
  ContentProviderResult[] results =
                          getContentResolver().applyBatch(AUTHORITY, ops);
} catch (OperationApplicationException e) {
  // Дії, які ви хочете прийняти, якщо операція провалюється
}
```

Закінчення рис. 4.48

Отримання та зміна даних за допомогою намірів

За допомогою відправлення намірів у додаток для роботи з контактами, які записано на пристрій, можна в обхід отримати доступ до постачальника контактів. Намір запускає інтерфейс користувача програми на пристрій, за допомогою якого користувач може працювати з контактами. Такий тип доступу дозволяє користувачеві виконувати такі дії:

вибір контактів зі списку та їхнє повернення в додаток для подальшої роботи;

зміна даних наявного контакту;

уставлення нового необробленого контакту для будь-яких інших акаунтів;

видалення контакту або його даних.

Якщо користувач уставляє або оновлює дані, можна спочатку зібрати ці дані, а потім відправити їх разом із наміром.

Під час використання намірів для доступу до постачальника контактів за допомогою додатка для роботи з контактами, наявного на пристрої, не потрібно створювати власний інтерфейс користувача або код для доступу до постачальника. Також не слід запитувати дозвіл на читання або запис у постачальник. Додаток для роботи з контактами, наявний на пристрої, може делегувати дозвіл на читання контакту, і оскільки вносять зміни в постачальник через інший додаток, не потрібен дозвіл на запис.

У табл. 4.7 далі наведено зведені відомості про операції, тип MIME та значення даних, які використовують для доступних завдань. Значення додаткових даних, які можна використовувати для `putExtra()`, указано в довідковій документації з `ContactsContract.Intents.Insert`.

Таблиця 4.7

Наміри в постачальнику контактів

Завдання	Дія	Дані	Типи MIME	Примітки
1	2	3	4	5
Вибір контакту зі списку	ACTION_PICK	Одне з наведеного: Contacts.CONTENT_URI (відображення списку контактів); Phone.CONTENT_URI (відображення номерів телефонів для необробленого контакту); StructuredPostal.CONTENT_URI (відображення списку поштових адрес для необробленого контакту); Email.CONTENT_URI (відображення адрес електронної пошти для необробленого контакту)	Не використовується	Відображення списку необроблених контактів або списку даних необробленого контакту (залежно від наданого URI контенту). Треба викликати метод startActivityForResult(), який повертає URI контенту для вибраного рядка. URI наведено у формі URI контенту таблиці, до якого додано LOOKUP_ID рядка. Додаток для роботи з контактами, установлений на пристрої, делегує цьому URI-контенту дозвіл на читання та запис, які діють протягом усього життєвого циклу операції. Додаткові відомості подані в статті Основні відомості про постачальника контенту
Уставлення нового необробленого контакту	Insert.ACTION	Н/Д	RawContacts.CONTENT_TYPE, тип MIME для набору необроблених контактів	Відображення екрана Додати контакт у додатку для роботи з контактами, який установлено на пристрої. Відображаються значення додаткових даних, доданих у намір під час відправлення за допомогою методу startActivityForResult() URI контенту нового доданого необробленого контакту передається назад у метод зворотного виклику onActivityResult() операції в аргументі Intent у полі data. Щоб отримати значення, викличте метод getData()

Закінчення табл. 4.7

1	2	3	4	5
Уставлення нового необробленого контакту	`Insert. ACTION`	Н/Д	`RawContacts. CONTENT_TYPE`, тип MIME для набору необроблених контактів	Відображення екрана *Додати контакт* у додатку для роботи з контактами, які установлено на пристрої. Відображаються значення додаткових даних, доданих у намір під час відправлення за допомогою методу `startActivityForResult()` URI контенту нового доданого необробленого контакту передається назад у метод зворотного виклику `onActivityResult()` операції в аргументі `Intent` у полі `data`. Щоб отримати значення, викличте метод `getData()`
Зміна контакту	`ACTION_EDIT`	`CONTENT_LOOKUP_URI` контакту. Операція редактора дозволить користувачеві змінити будь-які дані, пов'язані із цим контактом	`Contacts.CONTENT _ITEM_TYPE`, один контакт	Відображення екрана редагування контакту в додатку для роботи з контактами. Відображаються значення додаткових даних, доданих у намір. Коли користувач натискає на кнопку *Готово* для збереження внесених змін, операція повертається на передній план
Відображення засобу вибору, в якому також можна додавати дані	`ACTION INSERT OR_ EDIT`	Н/Д	`CONTENT ITEM_ TYPE`	Цей намір також відображає екран засобу вибору програми для роботи з контактами. Користувач може обрати контакт для зміни або додати новий контакт. Залежно від вибору, відображаються екран редагування або додавання контакту та додаткові дані, передані в намір. Якщо додаток відображає такі дані контакту, як адреса електронної пошти або номер телефону, скористайтеся цим наміром, щоб дозволити користувачеві додавати дані до наявного контакту. *Примітка.* Не потрібно відправляти значення назви в додаткові дані свого наміру, оскільки користувач завжди обирає наявну назву або додає нове. Крім того, якщо відправити назву, а користувач вирішить внести зміни, додаток для роботи з контактами відобразить відправлену назву, перезаписуючи попереднє значення. Якщо користувач не помітить цього та збереже зміни, старе значення буде втрачено

Додаток для роботи з контактами, наявний на пристрої, не дозволяє видаляти необроблені контакти або будь-які його дані за допомогою намірів. Замість цього для оброблення необробленого контакту слід використовувати метод `ContentResolver.delete()` або `ContentProviderOperation.newDelete()`.

Далі наведено фрагмент коду для створення та відправлення намірів, який уставляє новий необроблений контакт і його дані (рис. 4.49):

```
// Отримує значення із інтерфейсу користувача
String name = mContactNameEditText.getText().toString();
String phone = mContactPhoneEditText.getText().toString();
String email = mContactEmailEditText.getText().toString();

String company = mCompanyName.getText().toString();
String jobtitle = mJobTitle.getText().toString();

// Створює новий намір для відправлення в додаток контактів пристрою
Intent insertIntent = new
Intent(ContactsContract.Intents.Insert.ACTION);

// Установлює тип MIME
insertIntent.setType(ContactsContract.RawContacts.CONTENT_TYPE);

// Установлює нову назву контакту
insertIntent.putExtra(ContactsContract.Intents.Insert.NAME, name);

// Установлює нову компанію та посаду
insertIntent.putExtra(ContactsContract.Intents.Insert.COMPANY, company);
insertIntent.putExtra(ContactsContract.Intents.Insert.JOB_TITLE,
jobtitle);

/*
 * Демонструє додавання рядків даних як список масиву, пов'язаного
із ключем DATA
 */

// Визначає список масиву, щоб утримувати об'єкти ContentValues
для кожного рядка
ArrayList<ContentValues> contactData = new ArrayList<ContentValues>();
```

Рис. 4.49. **Уставлення нового необробленого контакту та його даних**

```
/** Визначає необроблений рядок контакту */

// Установлює рядок як об'єкт ContentValues
ContentValues rawContactRow = new ContentValues();

// Додає тип облікового запису та назву в рядок
rawContactRow.put(ContactsContract.RawContacts.ACCOUNT_TYPE,
mSelectedAccount.getType());
rawContactRow.put(ContactsContract.RawContacts.ACCOUNT_NAME,
mSelectedAccount.getName());

// Додає рядок до масиву
contactData.add(rawContactRow);

/*
 * Установлює рядок даних телефонного номера
 */

// Установлює рядок як об'єкт ContentValues
ContentValues phoneRow = new ContentValues();

// Указує тип MIME для цього рядка даних (усі рядки даних мають бути
зазначені за їхнім типом)
phoneRow.put(
        ContactsContract.Data.MIMETYPE,
        ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE
);

// Додає номер телефону та його тип до рядка
phoneRow.put(ContactsContract.CommonDataKinds.Phone.NUMBER, phone);

// Додає рядок у масив
contactData.add(phoneRow);

/*
 * Установлює рядок даних електронної пошти
 */

// Установлює рядок як об'єкт ContentValues
ContentValues emailRow = new ContentValues();
```

Продовження рис. 4.49

```
// Указує тип MIME для цього рядка даних (усі рядки даних мають бути
зазначені за їхнім типом)
emailRow.put(
        ContactsContract.Data.MIMETYPE,
        ContactsContract.CommonDataKinds.Email.CONTENT_ITEM_TYPE
);

// Додає адресу електронної пошти та тип рядка
emailRow.put(ContactsContract.CommonDataKinds.Email.ADDRESS, email);

// Додає рядок у масив
contactData.add(emailRow);


insertIntent.putParcelableArrayListExtra(ContactsContract.Intents.Insert
.DATA, contactData);

// Розсилає намір для старту додатка на пристрої та активності контакту
startActivity(insertIntent);
```

Закінчення рис. 4.49

Цілісність даних

Оскільки в репозиторії контактів міститься важлива та конфіденційна інформація, яка, як очікує користувач, правильна й актуальна, у постачальнику контактів передбачено чітко визначені правила для забезпечення цілісності даних. За зміни даних контактів зобов'язані дотримуватися цих правил. Ось найважливіші із цих правил:

1. Завжди слід додавати рядок `ContactsContract.CommonDataKinds.StructuredName` до кожного рядка, що додають `ContactsContract.RawContacts`.

2. Якщо в таблиці `ContactsContract.Data` є рядок `ContactsContract.RawContacts` без рядка `ContactsContract.CommonDataKi nds.StructuredName`, то можуть виникнути проблеми під час агрегування.

3. Завжди слід пов'язувати нові рядки `ContactsContract.Data` з їхніми батьківськими рядками `ContactsContract.RawContacts`.

4. Рядок `ContactsContract.Data`, який не пов'язаний із `ContactsContract.RawContacts`, не буде показано в додатку для роботи з контактами, що є на пристрої, а також може призвести до проблем з адаптерами синхронізації.

5. Слід змінювати дані тільки тих необроблених контактів, якими володіють.

6. Потрібно пам'ятати про те, що постачальник контактів, зазвичай, управляє даними з декількох акаунтів різних типів або послуг через Інтернет. Необхідно переконатися в тому, що додаток змінює або видаляє дані тільки для тих рядків, які належать вам, а також у тому, що він вставляє дані з використанням тільки тих типів та назв акаунту, якими управляєте.

7. Завжди слід використовувати для центрів, URI контенту, шляхів URI, назв стовпців, типів MIME та значень `TYPE` тільки ті константи, які визначені у класі `ContactsContract` і його підкласах.

Це дозволить уникнути помилок. Якщо яка-небудь константа застаріла, компілятор повідомить про це за допомогою відповідного попередження.

Рядки даних, що можна налаштувати

Створивши та використовуючи власні типи MIME, можна вставляти, змінювати, видаляти та витягувати в таблиці `ContactsContract.Data` власні рядки даних. Ці рядки обмежено використанням стовпчика, який визначено в `ContactsContract.DataColumns`, однак можна порівняти власні назви стовпців за типами рядків з назвами стовпців за замовчуванням. Дані для рядків відображаються в додатку для роботи з контактами, який записано на пристрій, проте їх не вдасться змінити або видалити, а також користувачі не зможуть додати додаткові дані. Щоб дозволити користувачам змінювати рядки даних, що можна налаштувати, необхідно реалізувати в додатку операцію редактора.

Для відображення даних, що можна налаштувати, слід указати файл `contacts.xml`, що містить елемент `<ContactsAccountType>` і один або декілька його `<ContactsDataKind>` дочірніх елементів. Додаткові відомості про це наведено в розділі `<ContactsDataKind> element`.

4.4.9. Адаптери синхронізації постачальника контактів

Постачальник контактів розроблено спеціально для оброблення операцій синхронізації даних контактів між пристроєм і службою в Інтернеті. Завдяки цьому користувачі можуть завантажувати наявні дані на новий пристрій і відправляти їх у новий обліковий запис. Синхронізація також забезпечує надання користувачеві завжди актуальних відомостей, незалежно від джерела їхнього додавання та внесених до них змін. Ще одна перевага синхронізації полягає в тому, що дані контактів доступні навіть у тому випадку, якщо пристрій не підключено до мережі.

Незважаючи на безліч різних способів реалізації синхронізації даних, у системі Android є платформа синхронізації, що підключається і дозволяє автоматизувати виконання таких завдань, як:

перевірка доступності мережі;

планування та виконання синхронізації на основі вподобань користувача;

перезапуск зупинених процесів синхронізації.

Для використання цієї платформи слід надати модуль адаптера синхронізації. Кожен адаптер синхронізації є унікальним для служби та постачальника контенту, проте здатний працювати з декількома обліковими записами в одній службі. У платформі також передбачено можливість використовувати кілька адаптерів синхронізації для однієї й тієї ж служби або постачальника.

Класи та файли адаптера синхронізації

Адаптер синхронізації реалізується як підклас класу `AbstractThreadedSyncAdapter` та встановлюється у складі додатка Android. Система дізнається про наявність адаптера синхронізації з елементів у маніфесті додатка, а також з особливого файла XML, на який є вказівка в маніфесті. У файлі XML визначають тип акаунту в онлайн-службі та центр постачальника контенту, які разом слугують унікальними ідентифікаторами адаптера. Адаптер синхронізації знаходиться в неактивному стані до тих пір, поки користувач не додасть тип акаунту і не включить синхронізацію з постачальником контенту. На цьому етапі система вступає в управління адаптером, за потреби викликаючи його для синхронізації даних між постачальником контенту та сервером.

Примітка. Використання типу акаунту для ідентифікації адаптера синхронізації дозволяє системі виявляти та групувати адаптери синхронізації, які звертаються до різних служб з однієї й тієї ж організації. Наприклад, у всіх адаптерів синхронізації для онлайн-служб Google один і той же тип акаунту – `com.google`. Коли користувачі додають на свої пристрої обліковий запис Google, усі встановлені адаптери синхронізації групуються разом; кожен з адаптерів синхронізовано тільки з окремим постачальником контенту на пристрої.

Оскільки в більшості служб користувачам спочатку необхідно підтвердити свою автентичність, перш ніж вони зможуть отримати доступ до даних, система Android пропонує платформу автентифікації, яка

аналогічна платформі адаптера синхронізації та часто використовується разом із нею. Платформа автентифікації використовує спільні структури перевірки автентичності, які становлять собою підкласи класу `AbstractAccountAuthenticator`. Така структура перевіряє справжність користувача таким чином:

1. Спочатку збирають відомості про назву користувача та його пароль або аналогічну інформацію (облікові дані користувача).
2. Потім облікові дані оговтуються у службу.
3. Нарешті, вивчається відповідь, отримана від служби.

Якщо служба прийняла облікові дані, структура перевірки автентичності може зберегти їх для використання в подальшому. Завдяки використанню структур перевірки автентичності, `AccountManager` може надати доступ до будь-яких маркерів автентифікації, які підтримує структура перевірки автентичності та які вона вирішує надати, наприклад, до маркерів автентифікації `OAuth2`.

Незважаючи на те що автентифікації не потребують, її використовують більшість служб для роботи з контактами. Проте не обов'язково використовувати платформу автентифікації Android для перевірки автентичності.

Реалізація адаптера синхронізації

Щоб реалізувати адаптер синхронізації для постачальника контактів, слід почати зі створення додатка Android, що містить такі компоненти:

1. Компонент `Service`, який відповідає на запити системи для прив'язування до адаптера синхронізації.
2. Коли системі потрібно запустити синхронізацію, вона викликає метод `onBind()` служби, щоб отримати об'єкт `IBinder` для адаптера синхронізації. Завдяки цьому система може викликати методи адаптера між процесами.

У *прикладі адаптера синхронізації* назвою класу цієї служби є `com.example.android.samplesync.syncadapter.SyncService`.

Безпосередньо адаптер синхронізації реалізовано як конкретний підклас класу `AbstractThreadedSyncAdapter`.

Цей клас не підходить для завантаження даних із сервера, відправлення даних із пристрою та вирішення конфліктів. Основну свою роботу адаптер виконує в методі `onPerformSync()`. Цей клас допускає створення тільки одного примірника.

У *прикладі адаптера синхронізації* адаптер визначено у класі `com.example.android.samplesync.syncadapter.SyncAdapter`.

Підклас класу `Application` є фабрикою для єдиного примірника адаптера синхронізації. Слід скористатися методом `onCreate()`, щоб створити екземпляр адаптера синхронізації, а потім надати статичний метод `get`, щоб повернути єдиний екземпляр у метод `onBind()` служби адаптера синхронізації.

Не обов'язково: компонент `Service`, який відповідає на запити системи для автентифікації користувачів.

`AccountManager` запускає службу, щоб почати процес автентифікації. Метод `onCreate()` служби створює екземпляр об'єкта структури перевірки автентичності. Коли системі потрібно запустити автентифікацію акаунту користувача для адаптера синхронізації додатка, вона викликає метод `onBind()` служби, щоб отримати об'єкт `IBinder` для структури перевірки автентичності. Завдяки цьому система може викликати методи структури перевірки автентичності між процесами.

У прикладі адаптера синхронізації назвою класу цієї служби є `com.example.android.samplesync.authenticator.AuthenticationService`.

Не обов'язково: конкретний підклас класу `AbstractAccountAuthenticator`, який обробляє запити на автентифікацію.

У цьому класі є методи, які `AccountManager` викликає для перевірки автентичності облікових даних користувача на сервері. Подробиці процесу автентифікації значно різняться, залежно від технології, що використовується на сервері. Щоб дізнатися докладніше про автентифікацію, слід звернутися до відповідної документації програмного забезпечення використовуваного сервера.

У *прикладі адаптера синхронізації* структуру перевірки автентичності визначено у класі – `com.example.android.samplesync.authenticator.Authenticator`.

Файли XML, у яких визначено адаптер синхронізації та структура перевірки автентичності для системи.

Описані раніше компоненти служби адаптера синхронізації та структури перевірки автентичності визначено в елементах `<service>` в маніфесті додатка. Ці елементи включають дочірні елементи `<meta-data>`, у яких є певні дані для системи.

Елемент `<meta-data>` для служби адаптера синхронізації вказує на файл XML `res/xml/syncadapter.xml`. У свою чергу, у цьому файлі задано

URI веб-служби для синхронізації з постачальником контактів, а також тип акаунту для цієї веб-служби.

Не обов'язково: елемент `<meta-data>` для структури перевірки автентичності вказує на файл XML `res/xml/authenticator.xml`. У свою чергу, у цьому файлі задано тип акаунту, який підтримує структуру перевірки автентичності, а також ресурси для інтерфейсу користувача, які відображаються у процесі автентифікації. Тип акаунту, зазначений у цьому елементі, має збігатися з типом акаунту, який задано для адаптера синхронізації.

4.5. Платформа доступу до сховища (Storage Access Framework)

Платформа доступу до сховища (Storage Access Framework, SAF) уперше з'явилася в Android версії 4.4 (API рівня 19). Платформа SAF полегшує користувачам пошук і відкриття документів, зображень та інших файлів у сховищах усіх постачальників, із якими вони працюють. Стандартний зручний інтерфейс дозволяє користувачам застосовувати єдиний для всіх додатків і постачальників спосіб пошуку файлів і доступу до останніх доданих файлів.

Хмарні або локальні служби зберігання можуть приєднатися до цієї екосистеми, реалізувавши клас `DocumentsProvider`, що інкапсулює їхні послуги. Клієнтські програми, яким потрібен доступ до документів постачальника, можуть інтегруватися із SAF за допомогою всього декількох рядків коду.

Платформа SAF містить такі компоненти:

постачальник документів – постачальник контенту, що дозволяє службі зберігання (наприклад, Диск Google) показувати файли, якими він управляє. Постачальник документів реалізується як підклас класу `DocumentsProvider`. Його схему засновано на традиційній файлової ієрархії, однак фізичний спосіб зберігання даних у постачальника документів залишається на розсуд розробника. Платформа Android містить кілька вбудованих постачальників документів, таких як *Завантаження*, *Зображення та Відео*;

клієнтську програму – користувальницький додаток, що викликає намір `ACTION_OPEN_DOCUMENT` і/або `ACTION_CREATE_DOCUMENT` і приймає файли, які повертаються постачальниками документів;

елемент вибору – системний, інтерфейс користувача, що забезпечує користувачам доступ до документів у всіх постачальників документів, які задовольняють критеріям пошуку, заданим у клієнтському додатку.

Платформа SAF серед інших надає такі функції:

дозволяє користувачам шукати контент у всіх постачальників документів, а не тільки в одній програмі;

забезпечує додатку можливість довготривалого, постійного доступу до документів, що належать постачальнику документів. Завдяки такому доступу, користувачі можуть додавати, редагувати, зберігати та видаляти файли, що зберігаються в постачальника;

підтримує декілька облікових записів і тимчасові кореневі каталоги, наприклад, постачальники на USB-накопичувачах, які з'являються, тільки коли накопичувач уставлений у порт.

4.5.1. Огляд

У центрі платформи SAF знаходиться постачальник контенту, який є підкласом класу `DocumentsProvider`. Усередині постачальника документів данні мають структуру традиційної файлової ієрархії (рис. 4.50):

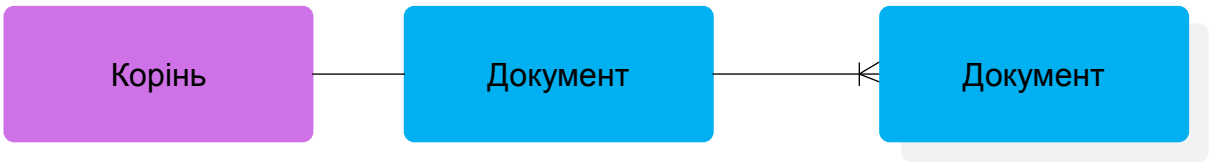

Рис. 4.50. **Традиційна файлова ієрархія**

На рис. 4.50 проілюстрована модель даних постачальника документів. На малюнку Root (Кореневий каталог) вказує на один об'єкт Document (Документ), який потім розгалужується в ціле дерево.

Слід звернути увагу на таке:

кожен постачальник документів надає один або кілька *кореневих каталогів*, що є відправними точками під час обходу дерева документів. Кожен кореневий каталог має унікальний ідентифікатор `COLUMN_ROOT_ID` і вказує на документ (каталог), що надає вміст на рівні, нижчому за кореневий. Кореневі каталоги динамічні за своєю конструкцією, щоб забезпечувати підтримку таких варіантів використання, як кілька облікових записів, тимчасові сховища на USB-накопичувачах і можливість для користувача увійти в систему та вийти з неї;

у кожному кореневому каталозі знаходиться один документ. Цей документ указує на кількість документів N, кожен із яких, у свою чергу, може вказувати на один або N документів;

кожен сервер сховища показує окремі файли та каталоги, посилаючись на них за допомогою унікального ідентифікатора `COLUMN_DOCUMENT_ID`. Ідентифікатори документів мають бути унікальними та не мінятися після надання, оскільки вони використовуються для видачі постійних URI, що не залежать від перезавантаження пристрою;

документ – це або файл, що відкривається (має конкретний MIME-тип), або каталог, що містить інші документи (із MIME-типом `MIME_TYPE_DIR`);

кожен документ може мати різні властивості, описувані такими прапорцями `COLUMN_FLAGS`, як `FLAG_SUPPORTS_WRITE`, `FLAG_SUPPORTS_DELETE` і `FLAG_SUPPORTS_THUMBNAIL`. Документ з одним і тим же ідентифікатором `COLUMN_DOCUMENT_ID` може знаходитися в декількох каталогах.

4.5.2. Потік управління

Як було сказано раніше, модель даних постачальника документів засновано на традиційній файлової ієрархії. Однак фізичний спосіб зберігання даних залишається на розсуд розробника за умови, що до них можна звертатися через API-інтерфейс `DocumentsProvider`. Наприклад, можна використовувати для даних хмарне сховище на основі тегів.

На рис. 4.51 наведено приклад того, як додаток для оброблення фотографій може використовувати SAF для доступу до збережених даних:

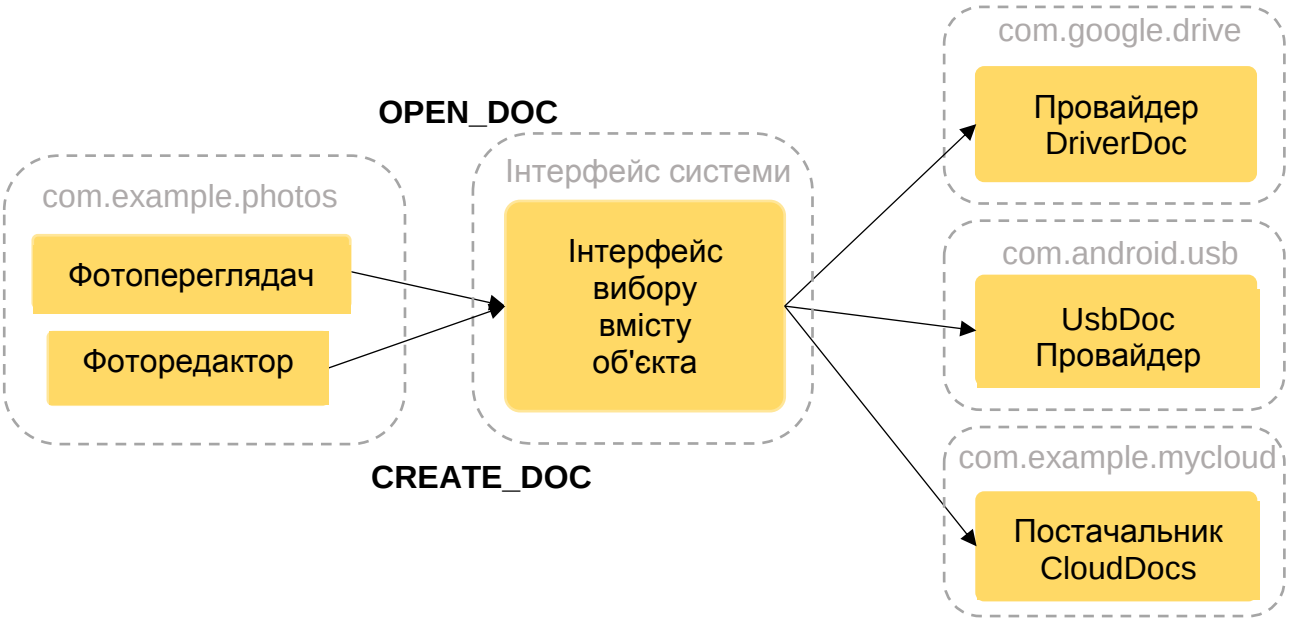

Рис. 4.51. **Потік управління Storage Access Framework**

Слід звернути увагу на таке:

на платформі SAF постачальники та клієнти не взаємодіють безпосередньо. Клієнт запитує дозвіл на взаємодію з файлами (тобто на читання, редагування, створення або видалення файлів);

взаємодія починається, коли додаток (у нашому прикладі обробляє фотографії) активізує намір `ACTION_OPEN_DOCUMENT` або `ACTION_CREATE_DOCUMENT`. Намір може містити фільтри для уточнення критеріїв, наприклад *надати відкриватися файлам із MIME-типом image*;

коли намір спрацьовує, системний елемент вибору переходить до кожного зареєстрованого постачальника та показує користувачеві кореневі каталоги з контентом, відповідним запитом;

елемент вибору надає користувачеві стандартний інтерфейс, навіть якщо постачальники документів значно різняться. Як приклад на рис. 4.51 зображено *Диск Google*, постачальник на USB-накопичувачі та хмарний постачальник.

На рис. 4.52 показано елемент вибору, у якому користувач для пошуку зображень вибрав обліковий запис *Диск Google*:

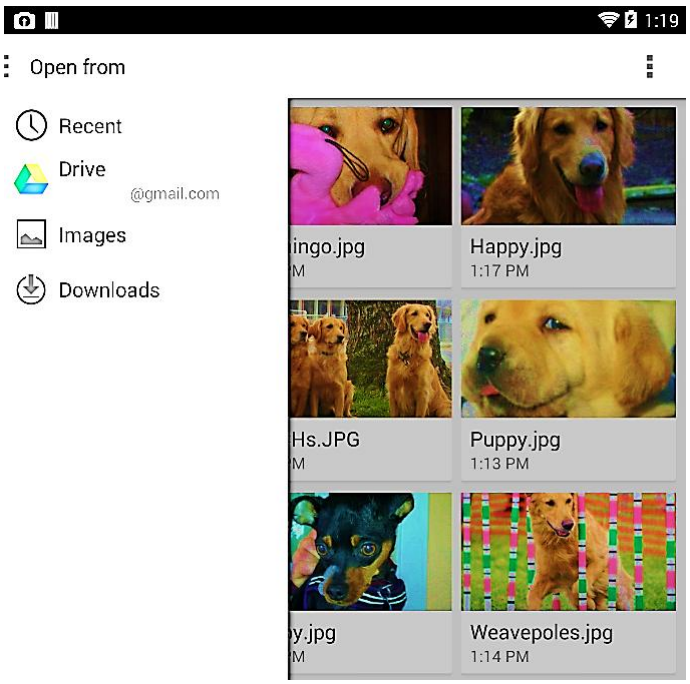

Рис. 4.52. **Елемент вибору**

Коли користувач вибирає *Диск Google*, зображення відображаються, як показано на рис. 4.53 із цього моменту користувач може взаємодіяти з ними будь-якими способами, які підтримуються постачальником і клієнтським додатком.

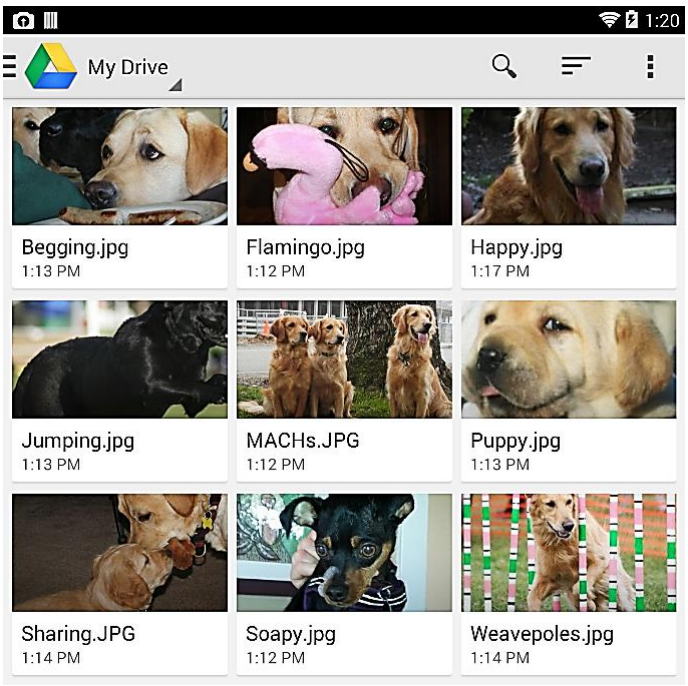

Рис. 4.53. **Зображення**

4.5.3. Створення клієнтської програми

В Android версії 4.3 і нижчих для того, щоб додаток міг отримувати файл від іншого додатка, він має активізувати намір, наприклад, `ACTION_PICK` або `ACTION_GET_CONTENT`. Після цього користувач має вибрати будь-який один додаток, щоб отримати файл, а додаток має надати користувачеві інтерфейс, за допомогою якого той зможе вибирати та отримувати файли.

Починаючи з Android 4.4 і вищих, у розробника є додаткова можливість – намір `ACTION_OPEN_DOCUMENT`, що відображає інтерфейс користувача елемента вибору, керованого системою. Цей елемент надає користувачеві огляд усіх файлів, доступних в інших додатках. Завдяки цьому єдиному інтерфейсу, користувач може вибрати файл у будь-якому з підтримуваних додатків.

Намір `ACTION_OPEN_DOCUMENT` не є заміною для наміру `ACTION_GET_CONTENT`. Розробнику слід використовувати те, що краще відповідає потребам програми:

`ACTION_GET_CONTENT`, коли програму потрібно просто прочитати або імпортувати дані. За такого підходу додаток імпортує копію даних, наприклад файл із зображенням.

`ACTION_OPEN_DOCUMENT`, коли програмі потрібна можливість довготривалого, постійного доступу до документів, що належить постачальнику документів. Як приклад можна назвати редактор фотографій, що дозволяє користувачам обробляти зображення, які зберігаються в постачальника документів.

У цьому підрозділі показано, як написати клієнтську програму, що використовує наміри `ACTION_OPEN_DOCUMENT` і `ACTION_CREATE_DOCUMENT`.

Пошук документів

У даному фрагменті коду намір `ACTION_OPEN_DOCUMENT` використовують для пошуку постачальників документів, що містять файли зображень (рис. 4.54):

```
private static final int READ_REQUEST_CODE = 42;
 ...
/** Запускаєте намір розкрутити вибір файлу UI і виберіть зображення. */
public void performFileSearch () {

  // ACTION_OPEN_DOCUMENT намір вибрати файл через файл системи браузер
  Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);

  /* Фільтр тільки показує результати, що можуть бути "відкриті", такі
як файл (на відміну від списку контактів або часових поясів) */
  intent.addCategory (Intent.CATEGORY_OPENABLE);

  // Фільтр для відображення тільки зображень, використовуючи тип MIME
даних зображень. Якщо хтось хоче знайти Ogg Vorbis файлів, тип буде "ау-
діо OGG". Для пошуку всіх документів, доступних через установлений про-
вайдер зберігання, це було б /*.
  intent.setType ("image/*");
  startActivityForResult (intent, READ_REQUEST_CODE);
}
```

Рис. 4.54. **Пошук постачальників документів, що містять файли зображень**

Слід звернути увагу на таке:

коли додаток активізує намір `ACTION_OPEN_DOCUMENT`, він запускає елемент вибору, що відображає всіх постачальників документів, що відповідають заданим критеріям;

додавання категорії `CATEGORY_OPENABLE` у *фільтри намірів* приводить до відображення тільки тих документів, які можна відкрити, наприклад, файлів із зображеннями;

оператор `intent.setType("image/*")` виконує подальшу фільтрацію, щоб відображалися тільки документи з MIME-типом image.

Оброблення результатів

Коли користувач вибирає документ в елементі вибору, він викликає метод `onActivityResult()`. Ідентифікатор URI, який указує на обраний документ, міститься в параметрі `resultData`. Щоб витягти URI, слід викликати `getData()`. Цей URI можна використовувати для отримання документа, потрібного користувачеві. Наприклад (рис. 4.55):

```
@Override
public void onActivityResult(int requestCode, int resultCode,
        Intent resultData) {

    // The ACTION_OPEN_DOCUMENT intent was sent with the request code
    // READ_REQUEST_CODE. If the request code seen here does not match,
it's the
    // response to some other intent, and the code below should not run
at all.

    if (requestCode == READ_REQUEST_CODE && resultCode ==
Activity.RESULT_OK) {
        // The document selected by the user will not be returned in the
intent.
        // Instead, a URI to that document will be contained in the
return intent
        // provided to this method as a parameter.
        // Pull that URI using resultData.getData ().
        Uri uri = null;
        if (resultData ! = null) {
            uri = resultData.getData();
            Log.i(TAG, "Uri:" + uri.());
showImage(uri);
        }
    }
}
```

Рис. 4.55. **Отримання документа, потрібного користувачеві**

Вивчення метаданих документа

Маючи у своєму розпорядженні URI документа, розробник отримує доступ до його метаданих. У наступному фрагменті коду метадані документа, що визначено ідентифікатором URI, зчитуються та записуються в журнал (рис. 4.56):

```
public void dumpImageMetaData(Uri uri) {

  /* Запит, оскільки він належить тільки до одного документа, буде пове-
ртати тільки один рядок. Там немає необхідності фільтрувати, сортувати,
або вибирати поля, так як ми хочемо, щоб усі поля були для одного доку-
мента. */
  Cursor cursor = getActivity().getContentResolver()
                       .query(uri, null, null, null, null, null);
  try {
    // moveToFirst() повертає false, якщо курсор має рядки.
    if (cursor ! = null && cursor.moveToFirst()) {

      // Зверніть увагу, що називається "Display Name". Це
      // постачальник - специфічний і не обов'язково є назвою файла.
      String displayName = cursor.getString(
                  cursor.getColumnIndex(OpenableColumns.DISPLAY_NAME));
      Log.i(TAG, "Display Name:" + displayName);

      int sizeIndex = cursor.getColumnIndex(OpenableColumns.SIZE);
      // Якщо розмір невідомий, значення, що зберігається, дорівнює нулю.
      // Але оскільки INT не може бути порожнім у Java, поведінка
      // залежить від конкретної реалізації, який просто
      // "непередбачуваний". З тим правилом перевірити, якщо це нуль
      // перед призначенням на міжнар. Це буде часто траплятися:
      // API-інтерфейс для зберігання дозволяє видалені файли, чий розмір
      // не може бути локально відомим.
      String size = null;
      if (!cursor.isNull(sizeIndex)) {
        // З технічної точки зору, стовпець зберігає Int, але
        // cursor.getString() буде автоматично виконувати перетворення.
        size = cursor.getString(sizeIndex);
      } else {
        size = "Unknown";
      }
      Log.i(TAG, "Size:" + size);
    }
  } finally {
    cursor.close();
  }
}
```

Рис. 4.56. **Метадані документа, що визначено ідентифікатором URI, і які зчитуються та записуються в журнал**

Відкриття документа

Отримавши URI документа, розробник може відкривати його і загалом робити з ним усе, що завгодно.

Об'єкт растрових зображень. На рис. 4.57 подано приклад коду для відкриття об'єкта `Bitmap`:

```
private Bitmap getBitmapFromUri(Uri uri) throws IOException {
  ParcelFileDescriptor parcelFileDescriptor =
                      getContentResolver().openFileDescriptor(uri, "R");
  FileDescriptor fileDescriptor =
                      parcelFileDescriptor.getFileDescriptor();
  Bitmap image = BitmapFactory.decodeFileDescriptor(fileDescriptor);
  parcelFileDescriptor.close();
  return image;
}
```

Рис. 4.57. **Відкриття об'єкта `Bitmap`**

Слід звернути увагу, що не потрібно виконувати цю операцію в потоці інтерфейсу користувача. Її необхідно виконувати у фоновому режимі, за допомогою `AsyncTask`. Коли файл із растровим зображенням відкриється, його можна відобразити у віджеті `ImageView`.

Отримання об'єкта InputStream. Далі наведено приклад того, як можна отримати об'єкт `InputStream` за ідентифікатором URI. У цьому фрагменті коду рядки файла зчитуються в об'єкт рядкового типу (рис. 4.58):

```
private String readTextFromUri(Uri uri) throws IOException {
  InputStream inputStream = getContentResolver().openInputStream(uri);
  BufferedReader reader =
                  new BufferedReader(new InputStreamReader(inputStream));
  StringBuilder stringBuilder = new StringBuilder();
  String line;
  while ((line = reader.readLine()) ! = null) {
    stringBuilder.append(line);
  }
  fileInputStream.close();
  parcelFileDescriptor.close();
  return stringBuilder.toString();
}
```

Рис. 4.58. **Отримання об'єкту `InputStream` за ідентифікатором URI**

Створення нового документа

Додаток може створити новий документ у постачальнику документів, використовуючи намір `ACTION_CREATE_DOCUMENT`. Щоб створити файл, потрібно вказати в намірі MIME-тип та назву файла, а потім запустити його з унікальним кодом запиту. Про інше подбає платформа (рис. 4.59):

```
// Ось деякі приклади того, як ви могли б назвати цей метод.
// Перший параметр є типом MIME, а другий параметр є назва
// зі створюваного файла:
// CreateFile ("текст / звичайний", "foobar.txt")
// CreateFile ("зображення / PNG", "mypicture.png");
// Унікальний код запиту.
private static final int WRITE_REQUEST_CODE = 43;
...
private void createFile(String mimeType, String fileName) {
  Intent intent = new Intent(Intent.ACTION_CREATE_DOCUMENT);

  // Фільтр показує тільки результати, що можуть бути "відкритими",
  // наприклад, файл (на відміну від списку контактів або часових поясів)
  intent.addCategory(Intent.CATEGORY_OPENABLE);

  //Створення файла із запитуваним типом MIME.
  intent.setType(mimeType);
  intent.putExtra(Intent.EXTRA_TITLE, fileName);
  startActivityForResult(intent, WRITE_REQUEST_CODE);
}
```

Рис. 4.59. **Створення нового документу у постачальнику документів**

Після створення нового документа можна отримати його URI за допомогою методу `onActivityResult()`, щоб мати можливість записувати в нього дані.

Видалення документа

Якщо у розробника є URI документа, а об'єкт `Document.COLUMN_FLAGS` основних напрямів містить прапорець `SUPPORTS_DELETE`, то документ можна видалити. Наприклад (рис. 4.60):

```
DocumentsContract.deleteDocument(getContentResolver(), uri);
```

Рис. 4.60. **Видалення документа**

Редагування документа

Платформа SAF дозволяє редагувати текстові документи на місці. У наступному фрагменті коду активізовано намір `ACTION_OPEN_DOCUMENT`, а категорія `CATEGORY_OPENABLE` використовується, щоб відображалися тільки документи, які можна відкрити. Потім відбувається подальша фільтрація, щоб відображалися тільки текстові файли (рис. 4.61):

```
private static final int EDIT_REQUEST_CODE = 44;
/** Відкрити файл для запису та додати текст до нього. * /
private void editDocument() {
  // ACTION_OPEN_DOCUMENT - намір вибрати файл за допомогою системи
  // файловий браузер.
  Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);

  // Фільтр показує тільки результати, що можуть бути "відкриті",
  // такі як файл (на відміну від списку контактів або часових поясів).
  intent.addCategory(Intent.CATEGORY_OPENABLE);

  // Фільтр для відображення тільки текстових файлів
  intent.setType("Text / plain");

  startActivityForResult(intent, EDIT_REQUEST_CODE);
}
```

Рис. 4.61. **Активізація наміру `ACTION_OPEN_DOCUMENT`, використання `CATEGORY_OPENABLE` та фільтрація**

Далі з методу `onActivityResult()` (див. *Оброблення результатів*) можна викликати код для виконання редагування. У наступному фрагменті коду об'єкт `FileOutputStream`, отримано за допомогою об'єкта класу `ContentResolver`. За замовчуванням використовується режим запису. Рекомендовано запитувати мінімально необхідні права доступу, тому не потрібно запитувати читання/запис, коли програмі слід тільки записати файл (рис. 4.62):

```
private void alterDocument(Uri uri) {
  try {
    ParcelFileDescriptor pfd = getActivity()
                               .getContentResolver().
                               .openFileDescriptor(uri, "W");
    FileOutputStream fileOutputStream =
                           new FileOutputStream(pfd.getFileDescriptor());
    fileOutputStream.write(("Overwritten by MyCloud at" +
                         System.currentTimeMillis() + "\ N").getBytes());
    // Нехай провайдер документа знає, що ви зробили, закривши потік.
    fileOutputStream.close();
    pfd.close();
  } catch (FileNotFoundException e) {
    e.printStackTrace();
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Рис. 4.62. **Отримання об'єкту `FileOutputStream` за допомогою об'єкта класу `ContentResolver`**

Утримання прав доступу

Коли додаток відкриває файл для читання або запису, система надає йому URI-дозвіл на цей файл. Дозвіл діє аж до перезавантаження пристрою. Припустимо, що у графічному редакторі потрібно, щоб у користувача була можливість відкрити безпосередньо в цьому додатку останні п'ять зображень, які він редагував. Якщо він перезапустив пристрій, виникає необхідність знову відсилати його до системного елемента вибору для пошуку файлів. Очевидно, це далеко не ідеальний варіант.

Щоб уникнути такої ситуації, розробник може утримати права доступу, надані системою його додатка. Додаток фактично бере постійний URI-дозвіл, запропонований системою. У результаті користувач отримує безперервний доступ до файлів із програми, незалежно від перезавантаження пристрою (рис. 4.63):

```
final int takeFlags = intent.getFlags()
       & (Intent.FLAG_GRANT_READ_URI_PERMISSION
```

Рис. 4.63. **Отримання безперервного доступу до файлів із програми**

```
            | Intent.FLAG_GRANT_WRITE_URI_PERMISSION);
// Перевіряє найсвіжіші дані.
getContentResolver().takePersistableUriPermission(uri, takeFlags);
```

Закінчення рис. 4.63

Залишається один заключний крок. Можна зберегти останні URI-ідентифікатори, із якими працював додаток. Однак не виключено, що вони втратять актуальність, оскільки інший додаток може видалити або модифікувати документ. Тому слід завжди викликати `getContentResolver().taePersistableUriPermission()`, щоб отримувати актуальні дані.

4.5.4. Створення власного постачальника документів

Під час розроблення програми, що надає послуги зі зберігання файлів (наприклад служби зберігання в хмарі), можна надати доступ до файлів за допомогою SAF, написавши власний постачальник документів. У цьому підрозділі показано, як це зробити.

Маніфест

Щоб реалізувати власний постачальник документів, необхідно додати в маніфест додатка таку інформацію:

цільовий API-інтерфейс рівня 19 або вищого;

елемент `<Provider>`, у якому оголошено нестандартний постачальник сховища;

назву постачальника, тобто назву його класу з назвою пакета. Наприклад: `com.example.android.storageprovider.MyCloudProvider`;

назву центра постачальника, тобто назву пакета (у цьому прикладі – `com.example.android.storageprovider`) із типом постачальника контенту (`documents`). Наприклад, `com.example.android.storageprovider.documents`;

атрибут `android:exported`, установлений у значення `True`. Необхідно експортувати постачальник, щоб його було видно іншим додаткам;

атрибут `android:grantUriPermissions`, установлений у значення `True`. Цей параметр дозволяє системі надавати іншим програмам доступ до контента постачальника;

дозвіл `MANAGE_DOCUMENTS`. За замовчуванням постачальник доступний усім. Додавання цього дозволу в маніфест робить постачальник доступним тільки системі. Це важливо для забезпечення безпеки;

атрибут `android:enabled`, що має логічне значення, визначене у файлі ресурсів. Цей атрибут призначено для відключення постачальника на пристроях під управлінням Android версії 4.3 і нижчих. Наприклад: `android:enabled="@bool/atLeastKitKat"`. Крім включення цього атрибута в маніфест, необхідно зробити таке:

– у файл ресурсів `bool.xml`, що розташований у каталозі `res/values/`, додати рядок: `<bool name="AtLeastKitKat">false</ Bool>`;

– у файл ресурсів `bool.xml`, що розташований у каталозі `res/values-v19/`, додати рядок: `<bool name="AtLeastKitKat">true</bool>`;

фільтр наміру з дією `android.content.action.DOCUMENTS_PROVIDER`, щоб постачальник з'являвся в елементі вибору, коли система буде шукати постачальників.

Далі наведено уривки зі зразка маніфесту, що містить постачальник (рис. 4.64):

```
<manifest ... >
  <uses-sdk
    android:minSdkVersion="19"
    android:targetSdkVersion="19"/>
  <provider
    android:name="com.example.android.storageprovider.MyCloudProvider"
    android:authorities="com.example.android.storageprovider.documents"
    android:grantUriPermissions="true"
    android:exported="true"
    android:permission="Android.permission.MANAGE_DOCUMENTS"
    android:enabled="@bool/atLeastKitKat">
    <intent-filter>
      <action android:name="android.content.action.DOCUMENTS_PROVIDER" />
    </intent-filter>
  </provider>
  </application>
</manifest>
```

Рис. 4.64. **Уривки зі зразка маніфесту, що містить постачальник**

Підтримка пристроїв під управлінням Android версії 4.3 і нижчих. Намір `ACTION_OPEN_DOCUMENT` доступно тільки на пристроях з Android версії 4.4 і вищих. Якщо додаток має підтримувати `ACTION_GET_CONTENT`, щоб обслуговувати пристрої, що працюють під управлінням Android 4.3 і нижчих, необхідно відключити фільтр намірів `ACTION_GET_CONTENT` у маніфесті для

пристроїв з Android версії 4.4 і вищих. Постачальник документів і намір ACTION_GET_CONTENT слід уважати взаємовиключними. Якщо додаток підтримує їх одночасно, він буде з'являтися в інтерфейсі системного елемента вибору двічі, пропонуючи два різні способи доступу до збережених даних. Це заплутає користувачів.

Відключати фільтр намірів ACTION_GET_CONTENT на пристроях з Android версії 4.4 і вищих рекомендовано таким чином:

1. У файл ресурсів bool.xml, розташований у каталозі res/values/, додати такий рядок (рис. 4.65):

```
<bool name="AtMostJellyBeanMR2">true</bool>
```

Рис. 4.65. **Рядок для файлу bool.xml (res/values/)**

2. У файл ресурсів bool.xml, розташований у каталозі res/values-v19/, додати такий рядок (рис. 4.66):

```
<bool name="AtMostJellyBeanMR2">false</bool>
```

Рис. 4.66. **Рядок для файлу bool.xml (res/values-v19/)**

3. Додати *псевдонім операції* (рис. 4.67), щоб відключити фільтр намірів ACTION_GET_CONTENT для версій 4.4 (API рівня 19) і вищих.

```
<!-- Цей псевдонім активності додається таким чином, щоб фільтр
GET_CONTENT наміру міг бути відключений для збірки на рівні API 19
і вище. -->
<Activity-alias android:name="com.android.example.app.MyPicker"
        android:targetActivity="com.android.example.app.MyActivity"
        ...
        android:enabled="@Bool/atMostJellyBeanMR2">
    <Intent-filter>
      <action android:name="android.intent.action.GET_CONTENT" />
      <category android:name="android.intent.category.OPENABLE" />
      <category android:name="android.intent.category.DEFAULT" />
      <data android:mimeType="image/*" />
      <data android:mimeType="video/*" />
    </Intent-filter>
</Activity-alias>
```

Рис. 4.67. **Відключення фільтру намірів ACTION_GET_CONTENT**

Контракти

Переважно, під час створення нестандартного постачальника контенту, одним із завдань є реалізація класів-контрактів. Клас-контракт становить клас `public final`, у якому містяться: визначення констант для URI, назви стовпців, типів MIME та інших метаданих постачальника. Платформа SAF надає розробнику такі класи-контракти, так що йому не потрібно писати власні:

`DocumentsContract.Document`;

`DocumentsContract.Root`.

Наприклад, коли до постачальника документів приходить запит на документи або кореневий каталог, можна повертати в курсорі такі стовпчики (рис. 4.68):

```
private static final String[] DEFAULT_ROOT_PROJECTION =
        new String[] {Root.COLUMN_ROOT_ID, Root.COLUMN_MIME_TYPES,
        Root.COLUMN_FLAGS, Root.COLUMN_ICON, Root.COLUMN_TITLE,
        Root.COLUMN_SUMMARY, Root.COLUMN_DOCUMENT_ID,
        Root.COLUMN_AVAILABLE_BYTES,};

private static final String[] DEFAULT_DOCUMENT_PROJECTION = new
        String[] {Document.COLUMN_DOCUMENT_ID,
Document.COLUMN_MIME_TYPE,
        Document.COLUMN_DISPLAY_NAME, Document.COLUMN_LAST_MODIFIED,
        Document.COLUMN_FLAGS, Document.COLUMN_SIZE,};
```

Рис. 4.68. **Стовпці**

Створення підкласу класу DocumentsProvider

Наступним кроком в розробленні власного постачальника документів є створення підкласу абстрактного класу `DocumentsProvider`. Як мінімум, необхідно реалізувати такі методи:

`queryRoots()`;

`queryChildDocuments()`;

`queryDocument()`;

`openDocument()`.

Це єдині методи, реалізація яких строго обов'язкова, проте існує набагато більше методів, можливо які, теж доведеться реалізувати. Подробиці наведено в описі класу `DocumentsProvider`.

Реалізація методу queryRoots. Реалізація методу `queryRoots()` має повертати об'єкт `Cursor`, що вказує на всі кореневі каталоги постачальників документів, використовуючи стовпці, визначені в `DocumentsContract.Root`.

У наступному фрагменті коду параметр `projection` надає конкретні поля, потрібні для виклику об'єкта. У цьому коді створюється курсор, і до нього додається один рядок, що відповідає одному кореневого каталогу (каталогу верхнього рівня), наприклад, *Завантаження* або *Зображення*. Більшість постачальників має тільки один кореневий каталог. Однак ніщо не заважає мати кілька кореневих каталогів, наприклад, за наявності декількох облікових записів. У цьому випадку досить додати в курсор ще один рядок (рис. 4.69):

```
@Override
public Cursor queryRoots(String[] projection)
                    throws FileNotFoundException {

  // Створення курсора або із запитуваних полів або за замовчуванням
  // проекція якщо "проекція" дорівнює нулю.
  final MatrixCursor result =
            new MatrixCursor(resolveRootProjection(projection));

  // Якщо користувач не увійшов у систему, повертає порожній кореневий
  // курсор. Це усуває OUR постачальник зі списку цілком.
  if (!isUserLoggedIn()) {
    return result;
  }

  // Можна мати кілька коренів (наприклад, для декількох облікових
  // записів у той самий додаток) - просто додайте кілька рядків курсо-
ра.
  // Побудувати один рядок для кореня під назвою MyCloud.
  final MatrixCursor.RowBuilder row = result.newRow();
  row.add(Root.COLUMN_ROOT_ID, ROOT);
  row.add(Root.COLUMN_SUMMARY,
          getContext().getString(R.string.root_summary));

  // FLAG_SUPPORTS_CREATE означає, щонайменше, один каталог під коренем
  // опору створенню документів. FLAG_SUPPORTS_RECENTS означає,
```

Рис. 4.69. **Додавання в курсор ще одного рядка**

```
    // щонайменш, один каталог під коренем підтримує створення документа.
    // FLAG_SUPPORTS_SEARCH дозволяє користувачам шукати всі документи
    // додатка
    row.add(Root.COLUMN_FLAGS, Root.FLAG_SUPPORTS_CREATE
                             | Root.FLAG_SUPPORTS_RECENTS
                             | Root.FLAG_SUPPORTS_SEARCH);

    // COLUMN_TITLE – це коренева назва (наприклад, галерея, Drive).
    row.add(Root.COLUMN_TITLE, getContext().getString(R.string.title));

    // Цей ідентифікатор документа не може змінитися, як тільки його
    // розподілено
    row.add(Root.COLUMN_DOCUMENT_ID, getDocIdForFile(mBaseDir));

    // Дочірні типи MIME використовують для фільтрації коріння та наявні
    // тільки в корні користувачів, які містять потрібний тип десь у їхній
    // ієрархії файлів.
    row.add(Root.COLUMN_MIME_TYPES, getChildMimeTypes(mBaseDir));
    row.add(Root.COLUMN_AVAILABLE_BYTES, mBaseDir.getFreeSpace());
    row.add(Root.COLUMN_ICON, R.drawable.ic_launcher);

    return result;
}
```

Закінчення рис. 4.69

Реалізація методу `queryChildDocuments`. Реалізація методу `queryChildDocuments()` має повертати об'єкт `Cursor`, що вказує на всі файли в заданому каталозі, використовуючи стовпці, визначені в `DocumentsContract.Document`.

Цей метод викликається, коли в інтерфейсі елемента вибору користувач вибирає кореневий каталог додатки. Метод отримує документи-нащадки каталогу на рівні, нижчому від кореневого. Його можна викликати на будь-якому рівні файлової ієрархії, а не тільки в кореневому каталозі. У наступному фрагменті коду створено курсор із запитаними стовпцями. Потім в нього заносять інформацію про кожного найближчого нащадка батьківського каталогу. Нащадком може бути зображення, ще один каталог, загалом будь-який файл (рис. 4.70):

```
@Override
public Cursor queryChildDocuments(String parentDocumentId,
          String[] projection,
          String sortOrder) throws FileNotFoundException {
  final MatrixCursor result = new
                    MatrixCursor(resolveDocumentProjection(projection));
  final File parent = getFileForDocId(parentDocumentId);
  for (File file : parent.listFiles()) {
    // Додає назву файлу відображення, MIME тип, розмір і так далі.
    includeFile(result, null, file);
  }
  return result;
}
```

Рис. 4.70. **Реалізація методу** `queryChildDocuments`

Реалізація методу `queryDocument`**.** Реалізація методу `queryDocument()` має повертати об'єкт `Cursor`, що вказує на заданий файл, використовуючи стовпці, визначені в `DocumentsContract.Document`.

Метод `queryDocument()` повертає ту ж інформацію, яку повертав `queryChildDocuments()`, але для конкретного файла (рис. 4.71):

```
@Override
public Cursor queryDocument(String documentId, String[] projection)
                        throws FileNotFoundException {
  // Створює курсор із запитуваною проекцією (проекцією за замовчуванням)
  final MatrixCursor result =
                new MatrixCursor(resolveDocumentProjection(projection));
  includeFile(result, documentId, null);

  return result;
}
```

Рис. 4.71. **Реалізація методу** `queryDocument()`

Реалізація методу `openDocument`**.** Необхідно реалізувати метод `openDocument()`, який повертає об'єкт `ParcelFileDescriptor`, що надає зазначений файл. Інші додатки зможуть скористатися поверненням об'єкта `ParcelFileDescriptor` для організації потоку даних. Система викликає цей метод, коли користувач вибирає файл, і клієнтський додаток запитує

доступ до нього, викликаючи метод `openFileDescriptor()`. Наприклад (рис. 4.72):

```
@Override
public ParcelFileDescriptor openDocument(final String documentId,
                                         final String mode,
                                         CancellationSignal signal)
            throws FileNotFoundException {

  Log.v(TAG, "OpenDocument, mode:" + mode);
  // Це нормальна робота мережевої операції в цьому методі,
  // щоб завантажити документ, до тих пір, поки ви періодично перевіряєте
  // CancellationSignal. Якщо у вас є дуже великий файл для передачі
  // по мережі, кращим рішенням можуть бути труби або розетки
  // (див ParcelFileDescriptor для допоміжних методів).
  final File file = getFileForDocId(documentId);
  final boolean isWrite = (mode.indexOf('W') ! = -1);
  if(isWrite) {
    // Прикріплення близького слухача, якщо документ відкритий у режимі
    // запису.
      try {
        Handler handler = new Handler(getContext().getMainLooper());
        return ParcelFileDescriptor.open(file, accessMode, handler,
          new ParcelFileDescriptor.OnCloseListener() {
            @Override
            public void onClose(IOException e) {
               Log.i(TAG, "A file with id" +
                          documentId + "Has been closed! Time to" +
                          "Update the server.");
            }
          });
      } catch (IOException e) {
        throw new FileNotFoundException("Failed to open document with id"
              + documentId + "And mode" + mode);
      }
    } else {
      return ParcelFileDescriptor.open(file, accessMode);
    }
  }
}
```

Рис. 4.72. **Реалізація методу** `openFileDescriptor()`

Безпека

Припустімо, що постачальник документів становить захищену паролем службу зберігання у хмарі, а додаток має переконатися, що ввійшли в систему, перш ніж він надасть йому доступ до файлів. Які заходи має вжити додаток, якщо користувач не виконав вхід? Рішення полягає в тому, щоб реалізація методу `queryRoots()` не повертала кореневих каталогів. Інакше кажучи, це має бути порожній кореневої курсор (рис. 4.73):

```
public Cursor queryRoots(String[] projection)
          // throws FileNotFoundException {
  ...
  // Якщо користувач не ввійшов у систему, повертає порожній кореневий
  // курсор. Це усуває OUR постачальник зі списку цілком.
  if (!isUserLoggedIn()) {
    return result;
}
```

Рис. 4.73. **Порожній кореневий курсор**

Наступний крок полягає у виклику методу `getContentResolver().notifyChange()`.

Пам'ятайте об'єкт `DocumentsContract`. Слід скористатися ним для створення відповідного URI. У наступному фрагменті коду систему сповіщають про необхідність опитувати кореневі каталоги постачальника документів, коли змінюється статус входу користувача в систему. Якщо користувач не виконав вхід, метод `queryRoots()` поверне порожній курсор, як показано раніше. Це гарантує, що документи постачальника будуть доступні тільки користувачам, що ввійшли в постачальник (рис. 4.74):

```
private void onLoginButtonClick() {
  loginOrLogout();
  getContentResolver().notifyChange(
                       DocumentsContract.buildRootsUri(AUTHORITY),
                       null);
}
```

Рис. 4.74. **Реалізація методу `onLoginButtonClick()`**

Запитання для самодіагностики

1. Назвіть основні типи сховищ для даних, доступних мобільному додатку на платформі Android.

2. Призначення постачальника контактів.

3. Локальна база даних SQLite.

4. Опишіть процес отримання даних за допомогою завантажувачів (`Loaders`).

5. Опишіть можливості доступу до вбудованих баз даних на прикладі *Постачальника контактів*.

6. Укажіть можливості платформи доступу до сховищ SAF.

Рекомендована та використана література

1. Барабанова М. И. Экономико-математическая модель динамики дохода отрасли связи России / М. И. Барабанова, В. П. Воробьев, В. Ф. Минаков // Известия Санкт-Петербургского государственного экономического университета. – 2013. – № 4 (82). – С. 24–28.

2. Голощапов А. Google android. Системные компоненты и сетевые коммуникации / А. Голощапов. – Санкт-Петербург : БХВ-Петербург, 2012. – 384 с.

3. Голощапов А. Google android. Создание приложений для смартфонов и планшетных ПК / А. Голощапов. – Санкт-Петербург : БХВ-Петербург, 2013. – 832 с.

4. Гриффитс Д. Head first. Программирование для android / Д. Гриффитс, Д. Гриффитс. – Санкт-Петербург : Питер, 2016. – 704 с.

5. Макарчук Т. А. Облачные решения построения информационных систем управления ресурсами организации / Т. А. Макарчук, В. Ф. Минаков, В. А. Щугорева // Международный научно-исследовательский журнал = Research Journal of International Studies. – 2014. – № 1-1 (20). – С. 68–69.

6. Минаков В. Ф. Информационное общество и проблемы прикладной информатики / В. Ф. Минаков, Т. Е. Минакова // Международный научно-исследовательский журнал = Research Journal of International Studies. – 2014. – № 1-1 (20). – С. 69–70. Nauka-rastudent.ru. – 2015. – No. 02 (014-2015).

7. Саати Т. Принятие решений. Метод анализа иерархий / Т.Саати. – Москва : Радио и связь. – 1993. – 278 с.

8. Хашими С. Разработка приложений для Android / С. Хашими, С. Коматинени, Д. Маклин. – Санкт-Петербург : Питер, 2011. – 736 с.

9. Бордюг В. Л. Выбор инструментов для разработки мобильного приложения методом анализа иерархии Т. Саати [Электронный ресурс] / В. Л. Бордюг, Е. Г. Панченко, О. Н. Трифонова // Nauka-rastudent.ru. – 2015. – No. 02 (014-2015) – Режим доступа : URL: http://nauka-rastudent.ru/14/2419.

10. AIDE-IDE for Android Java C++ [Electronic resource]. – Access mode : https://play.google.com/store/apps/details?id=com.aide.ui 16.06.2015.

11. Android App Stats [Electronic resource]. – Access mode : http://www.androlib.com/appstats.aspx 16.06.2015.

12. Android – Invoke JNI Based Methods (Bridging C/C++ And Java) [Electronic resource]. – Access mode : https://davanum.wordpress.com/2007/12/09/android-invoke-jni-based-methods-bridging-cc-and-java/ 16.06.2015.

13. Android NDK [Electronic resource]. – Access mode : https://developer.android.com/tools/sdk/ndk/index.html 16.06.2015.

14. Android Studio [Electronic resource]. – Access mode : http://developer.android.com/sdk/index.html 16.06.2015.

15. AndroWish [Electronic resource]. – Access mode : http://www.androwish.org/index.html/home 18.06.15.

16. App Components [Electronic resource]. – Access mode : https://developer.android.com/guide/components/index.html 21.10.2016.

17. App Inventor Book, Classic version [Electronic resource]. – Access mode : http://www.appinventor.org/book.

18. Apps for Programming on Android [Electronic resource]. – Access mode : http://android.appstorm.net/roundups/developer/15-apps-for-programming-on-android/ 18.06.2015.

19. Backup & restore Android apps using adb [Electronic resource]. – Access mode : http://jonwestfall.com/2009/08/backup-restore-android-apps-using-adb.

20. Dalvik Executable format [Electronic resource]. – Access mode : https://source.android.com/devices/tech/dalvik/dex-format.html 16.06.2015.

21. Documenting APIs with Examples: Lessons Learned with the API Miner Platform [Electronic resource] / João Eduardo Montandon, Hudson Borges, Daniel Felix, Marco Tulio Valente. – Access mode : http://homepages.dcc.ufmg.br/~mtov/pub/2013_wcre_apiminer.pdf.

22. Google Play Hits 1 Million Apps [Electronic resource]. – Access mode : http://mashable.com/2013/07/24/google-play-1-million/ 16.06.2015.

23. Google: 3 Billion Android Apps Installed; Downloads Up 50 Percent From Last Quarter [Electronic resource]. – Access mode : http://techcrunch.com/2011/04/14/google-3-billion-android-apps-installed-up-50-percent-from-last-quarter/ 16.06.2015.

24. How to Write a Simple Application [Electronic resource]. – Access mode : https://code.google.com/p/simple/wiki/HowToWriteASimpleApplication 18.06.15.

25. Introduction to Android [Electronic resource]. – Access mode : https://developer.android.com/guide/index.html 21.10.2016.

26. Java Editor [Electronic resource]. – Access mode : https://play.google.com/store/apps/details?id=air.Java Editor 16.06.2015.

27. JavaIDEdroid [Electronic resource]. – Access mode : https://play.google.com/store/apps/details?id=ch.tanapro.JavaIDEdroid 16.06.2015.

28. List of IDEs for Android App Development, Which is Best for You? [Electronic resource]. – Access mode : http://tekeye.biz/2014/list-of-android-app-development-ides 18.06.2015.

29. Native C applications for Android [Electronic resource]. – Access mode : http://benno.id.au/blog/2007/ 11/13/android-native-apps 16.06.2015.

30. NBAndroid [Electronic resource]. – Access mode : http://plugins.netbeans.org/plugin/19545/nbandroid 16.06.2015.

31. Programming Made Simple [Electronic resource]. – Access mode : https://code.google.com/p/simple/ 18.06.2015.

32. RAD Studio XE8 [Electronic resource]. – Access mode : https://www.embarcadero.com/products/rad-studio.

33. SDK Tools [Electronic resource]. – Access mode : http://developer.android.com/tools/sdk/tools-notes.html 16.06.2015.

34. SKIA graphics library in chrome: first impressions [Electronic resource]. – Access mode : http://www. atoker.com/blog/2008/09/06/skia-graphics-library-in-chrome-first-impressions/ 16.06.2015.

35. Smartphone OS Market Share, Q1 2015 [Electronic resource]. – Access mode : http://www.idc.com/prodserv/smartphone-os-market-share.jsp 16.06.2015.

36. The Perfect Platform for Game Developers: Android [Electronic resource]. – Access mode : http://www. developer.com/ws/android/client/the-perfect-platform-for-game-developers-android.html 18.06.2015.

37. The Professional Android IDE [Electronic resource]. – Access mode : http://www.jetbrains.com/idea/features/android.html 16.06.2015.

38. Tools Overview [Electronic resource]. – Access mode : http://developer.android.com/tools/help/index.html 16.06.2015.

39. Top 10 Android Apps and IDE for Java Coders and Programmers [Electronic resource]. – Access mode : https://blog.idrsolutions.com/2014/12/android-apps-ide-for-java-coder-programmers/ 18.06.2015.

40. User Interface [Electronic resource]. – Access mode : https://developer.android.com/guide/topics/ui/index. html 21.10.2016.

41. Visual C++ Cross-Platform Mobile [Electronic resource]. – Access mode : https://www.visualstudio.com/en-us/features/cplusplus-mdd-vs.aspx 18.06.2015.

42. What you should know about Hypernext Android Creator (HAC)? [Electronic resource]. – Access mode : https://sites.google.com/site/androidappdevsindia/what-you-should-know-about-hypernext-android-creator-hac.

Зміст

НАВЧАЛЬНЕ ВИДАННЯ

Поляков Андрій Олександрович
Федорченко Володимир Миколайович
Шматко Олександр Володимирович

АНАЛІЗ МЕТОДІВ І ТЕХНОЛОГІЙ РОЗРОБЛЕННЯ МОБІЛЬНИХ ДОДАТКІВ ДЛЯ ПЛАТФОРМИ ANDROID

Навчальний посібник

Самостійне електронне текстове мережеве видання

Відповідальний за видання *О. Г. Руденко*

Відповідальний редактор *М. М. Оленич*

Редактор *О. Г. Доценко*

Коректор *Т. А. Маркова*

План 2017 р. Поз. № 4-ЕНП. Обсяг 286 с.

Видавець і виготовлювач – ХНЕУ ім. С. Кузнеця, 61166, м. Харків, просп. Науки, 9-А

Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру
ДК № 4853 від 20.02.2015 р.