МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

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

АНАЛІЗ МЕТОДІВ І ТЕХНОЛОГІЙ РОЗРОБКИ МОБІЛЬНИХ ДОДАТКІВ ДЛЯ ПЛАТФОРМИ ANDROID. ЧАСТИНА 2

Навчальний посібник
для студентів спеціальності
121 «Інженерія програмного забезпечення»

Рекомендовано Вченою радою
Національного технічного університету
«Харківський політехнічний інститут»

Харків 2018

УДК 004.42(07)

Ш 33

Рецензенти:

О. Г. Руденко, д-р техн. наук, проф., Харківський національний економічний університет ім. С. Кузнеця;

С. В. Мінухін, д-р техн. наук, проф., Харківський національний економічний університет ім. С. Кузнеця

Рекомендовано Вченою радою НТУ «ХПІ» як навчальний посібник для студентів спеціальності «Інженерія програмного забезпечення», протокол №10 від 24.11.2017 р.

Шматко О. В.

Ш 33 Аналіз методів і технологій розробки мобільних додатків для платформи Android : навч. посіб. / О. В. Шматко, А. О. Поляков, В. М. Федорченко. – Харків : НТУ «ХПІ», 2018. – 284 с.

ISBN

Розкрито основи програмування для платформи Android. Розглянуто архітектуру платформи та операційної системи Android. Досліджено архітектуру мобільного програмного додатку. Розкрито сутність та призначення основних програмних компонентів: Activity, Service, Content Provider і Broatcast Resiver. Наведено методи побудови інуїтивного інтерфейсу користувача. Подано достатню кількість пояснювальних прикладів. Розглядаються різні форми збереження та обробки даних на мобільному пристрої.

Для студентів вищих навчальних закладів спеціальності 121 «Інженерія програмного забезпечення»

Іл. 48. Бібліогр. 42 назв

УДК 004.42(07)

ISBN 978-966-000-000-0

ЗМІСТ

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 інші). У таких умовах господарювання вимагає від фахівців з економічного управління всебічного використання новітніх інформаційних технологій. Широкі можливості мобільних засобів в питаннях збору, обробки та видачі необхідної інформації здатні значно підвищити якість економічних розрахунків, зробити більш ефективним процес обґрунтування економічних рішень.

Таким чином, процес розробки мобільних додатків стає актуальним напрямком у IT-індустрії. Сучасні компанії, такі, як Google, Apple, Microsoft та інші, розробили мобільні платформи, що включають мобільні операційні системи та засоби розробки (SDK, Software Developer Kit). Важливою особливістю мобільних пристроїв є те, що вони мають обмежене джерело живлення, невеликій розмір екрана та набір різноманітних сенсорів. Процес розробки мобільних додатків є достатньо технологічним і потребує певних компетенцій з об'єктно-орієнтованого програмування, знання SQL, проектування баз даних та UI, розуміння мережевої взаємодії, тестування програмного забезпечення.

Найбільш росповсюдженою мобільною платформою є Android. Вона складає понад 82 % від усіх засобів на 2016 р., що підтверджує її універсальніть і перспективність для подальшого вивчення. Після опанування матеріалу підручника студенти зможуть оволодіти такими компетенціями: проектувати та розробляти вимоги до мобільного додатка, з урахуванням особливостей мобільної платформи; визначати та класифікувати вимоги до розробки 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 даному посібнику розглядаються чотири теми, які слід вивчати поступово. У першому розділі подано основні положення мобільної платформи, розглянуті процеси та інструменти розробки для платформи Android, її архітектура. У другому розділі досліджуються базові компоненти для створення інтерфейсу користувача і базові макети. У третьому розділі досліджується робота зі службами, що дозволяє використовувати у додатку тривалі процеси обробки даних, що виконуються у фонових потоках та процесах. Четвертий розділ присвячений вивченню компонентів з обробки та збереження даних як локального пристрою, так Cloud Storage.

1. АРХІТЕКТУРА ТА КОМПОНЕНТИ МОБІЛЬНИХ ПЛАТФОРМ

1.1. Огляд операційного середовища Android

Платформа Android є розробкою групи Open Handset Alliance, яка поставила перед собою 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 розробки програмного забезпечення з відкритим кодом.

Першим телефоном, який просувався на ринок оператором T-Mobile і працював на платформі Android, був пристрій G1 від HTC. Цей телефон вийшов на ринок через рік після перших згадок про нього. Для розробки 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 в свій телефон. Всього 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 з великими функціональними можливостями. У підсистему інтерфейсу, призначеного для користувача, входять:

– вікна;

– подання;

– віджети для відображення загальних елементів, таких, як поля, що редагуються, списки та списки, що розгортаються.

В Android вбудовано браузер з движком WebKit з 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 використовуються сервіси, які надаються Google; наприклад, в своїх прикладних програмах можна посилатися на Google Maps для позиціонування пристрою. В стек про-

грамного забезпечення Android входить також підтримка сервісів, заснованих на визначенні місця розташування (наприклад, GPS), та акселерометрів, хоча не усі пристрої на цій платформі оснащені необхідним обладнанням. Присутня також підтримка відеокамери.

Через обмеження ресурсів мобільні пристрої завжди мали великі проблеми з обробкою графіки/мультимедіа та способами зберігання даних, на відміну від персональних комп'ютерів. Для вирішення даної проблеми платформа Android пропонує вбудовану підтримку 2-D і 3-D графіки, з використанням бібліотеки OpenGL. Для забезпечення зберігання даних платформа Android використовує популярну базу даних з відкритим вихідним кодом SQLite. На рис. 2.1 зображена спрощена схема рівнів програмного забезпечення Android.

Рисунок 1.1 – Рівні програмного забезпечення Android

1.3. Огляд віртуальних машин Android

1.3.1. Віртуальна машина Dalvik

Операційна система Android працює поверх ядра Linux. Для створення Android-додатків спочатку використовувалася мова програмування Java, а потім виконувалися прикладні програми у віртуальній машині (VM). Необхідно звернути увагу на те, що віртуальна машина – це не віртуальна машина Java (JVM), а відкрита технологія Dalvik Virtual Machine. При запуску програми Android створюється і запускається окремий екземпляр Dalvik VM, який, в свою чергу, розташований в межах керованого ядром Linux процесу, як показано на рис. 1.2.

Рисунок 1.2 – Dalvik VM

Dalvik є віртуальною машиною в межах операційної системи Android від Google, яка виконує прикладні програми, що 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 використовуються в мобільних пристроях, таких, як мобільні телефони та планшетні комп'ютери, а останнім часом на таких пристроях, як смарт-телевізори. Dalvik є програмним забезпеченням з відкритим вихідним кодом, написаним Dan Bornstein, який назвав його на честь рибальського селища Dalvik в Ісландії.

Програми для Android, як правило, 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` (оптимізований Dalvik Executable) файлах. Компактний формат Dalvik Executable призначений для систем, які обмежені з точки зору пам'яті та швидкості процесора.

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

1.3.2. Віртуальна машина ART

Android Runtime (ART) є середовищем виконання прикладних програм, яке використовується Android. ART виконує перетворення байт-коду програми в інструкції, які потім виконуються 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.

У той час як Dalvik інтерпретує частину в байт-коді прикладної програми, виконання ART цих коротких відрізків байт-коду, яке називається «слідом», забезпечує значне підвищення продуктивності.

На відміну від Dalvik, ART вводить використання ahead-of-time (AOT) компіляцію шляхом компіляції цілих прикладних програм в машинний код при їх установці. 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 самий вхідний байт-код, що і Dalvik, який надається через стандартні `.dex` файли як частина APK файлів, в той час як `.odex` файли замінюються Executable and Linkable Format (ELF) файлами (рис. 1.3).

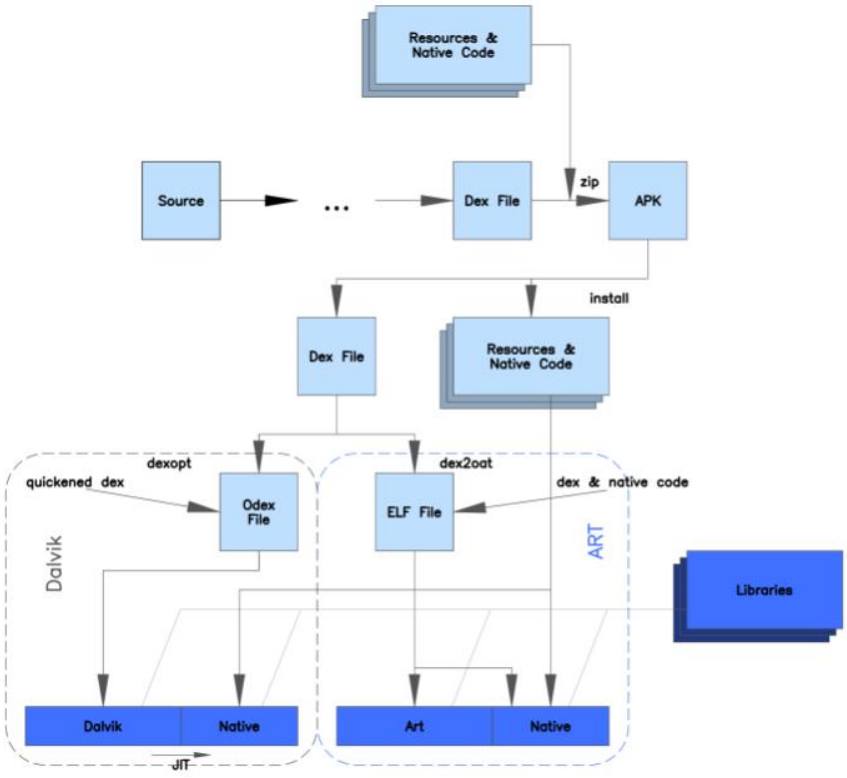

Рисунок 1.3 – Порівняння архітектури Dalvik та ART

Після того як прикладна програма буде зібрана та скомпільована на пристрої, утиліта dex2oat запускається з виконуваного ELF-файлу. У результаті ART усуває відмінності виконання прикладних програм, накладні витрати, пов’язані з інтерпретацією Dalvik, та трасування на основі JIT компіляції.

ART вимагає додаткового часу для компіляції при установці прикладної програми, а прикладні програми вимагають трохи більше місця для зберігання (як правило використовується флеш-пам’ять) скомпільованого коду.

В Android 4.4 «KitKat» з’явилася технологія попереднього огляду ART, в тому числі як альтернативного середовища виконання та збереження, замість Dalvik, як віртуальної машини поза вибором. У подальшому в Android 5.0 «Lollipop» Dalvik був повністю замінений на ART.

Android-прикладна програма містить елементи одного або декількох наступних типів.

Дії (Activities)

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

Сервіси (Services)

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

Джерела даних (Content providers)

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

Приймачі (Broadcast receivers)

Android-прикладна програма може запускатися для обробки елемента даних або для реагування на події, наприклад на отримання текстового повідомлення.

Прикладна програма для Android розгортається на пристрої разом з файлом AndroidManifest.xml. Цей файл містить необхідну інформацію про конфігурацію, яка дозволяє правильно встановити прикладну програму на пристрої. Він містить також необхідні імена класів та типи подій, які може обробляти прикладна програма, та дозволи, необхідні для її роботи. Так, якщо з прикладною програмою потрібен доступ до мережі (наприклад, щоб завантажити файл), відповідний дозвіл має бути явно вказаний у файлі-маніфесті. Цей конкретний дозвіл можуть мати багато прикладних програм. Такий захист шляхом декларування допомагає зменшити ймовірність пошкодження пристрою через некоректно написану прикладну програму.

1.4. Керування ресурсами мобільних пристроїв

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

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

Зміни, які вносяться до базової версії ядра, можна розділити на такі: виправлення помилок, засоби для підвищення простору для користувача (low memory killer, binder, ash mem, logger і т. д.), нова інфраструктура (особливо wake locks), підтримка нових SoCs (msm7k, msm8k і т. д.) та плат/пристроїв. В майбутньому специфічні ядра Android та базові ядра Linux повинні об'єднатися, але цей процес йде повільно і забере деякий час.

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

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

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

Інтерфейс користувача для створення блокіраторів використовує файл `/sys/power/wakelock`, в якому записується ім'я нового блокіратора. Для зняття блокування процес записує ім'я файлу в `/sys/power/wakeunlock`. Для блокування можна вказати час, після якого воно спрацює автоматично. Всі системи, які використовують на даний час сервіс блокування, вказуються у файлі `/proc/wakelocks`.

Інтерфейс ядра для блокіраторів пробудження дозволяє вказати, чи повинен `wakelock` запобігати низькому енергоспоживанню чи припиненню роботи системи. `WakeLock` створюється процесом `wakelockinit()` та вилучається `wakelockdestroy()`. Створений блокіратор може бути запущений процесом

wakelock() та розблокований wakeunlock(). Як і в просторі користувача, можна визначити тайм-аут для блокування. Концепція блокіраторів глибоко інтегрована 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 керування живленням. PowerManager забезпечує чотири різних види блокування:

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

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

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

Anonymous SHared MEMory забезпечує іменовані блоки пам'яті, які можуть бути поділені 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 був написаний 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) та використовує евристичні аналізатори для визначення «шкідливості» процесу, щоб мати можливість припинити його роботу. Вибираються процеси з найбільшою кількістю комірок з малою кількістю пам'яті. Така поведінка може бути незручною для користувача, оскі-

льки `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. Він виконує 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

У той час як більшість Android-прикладних програм написані мовою Java, є багато відмінностей між API Java та Android API. Основною відмінністю є те, що Android не використовує віртуальну машину Java, замість неї використовуються дві інші: 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 її від інших стандартних віртуальних машин: VM було розроблено, щоб використовувати менше пам'яті.

Пул регістрів був змінений, щоб використовувати тільки 32-бітові індекси для спрощення інтерпретатора.

Стандартний Java байт-код виконує 8-розрядні команди стека. Локальні змінні величини повинні 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 в 4 біти «віртуального регістра».

Оскільки байт-код, який завантажується віртуальною машиною Dalvik відрізняється від байт-коду Java Virtual Machine, та в зв'язку з певним способом завантаження класів віртуальною машиною Dalvik, відсутня можливість завантажувати jar-файли. Інший порядок також повинен бути використаний для завантаження Android-бібліотеки. При цьому зміст базового DEX-файлу має бути скопійовано на карту пам'яті перед завантаженням.

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 не використовується 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;

– властивість «`User.dir`» має інше значення на Android;

– властивостей «`User.home`» та «`User.name`» не існує в Android.

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.

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

Графіка та бібліотека віджетів. Android не використовує Abstract Window Toolkit та бібліотеку Swing. Інтерфейс користувача побудований з використанням подань об'єктів. Android використовує фреймворк, подібний до `Swing`, а не `JComponents`. Проте, Android-віджети не є `JavaBeans`.

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

Платформа Android підтримує відносно велику підмножину бібліотек Java Standard Edition 5.0. Деякі функції були видалені, 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 виключень;
- 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 субпакетм) – 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).

Існує ряд сторонніх бібліотек для Android SDK:

- org.apache.commons.codec – утиліти для кодування та декодування;

- org.apache.commons.httpclient – HTTP-аутентифікація, cookies, методи та протоколи;
- org.bluez – підтримка Bluetooth;
- org.json – формат JavaScript Object Notation.

1.6. Огляд архітектурних моделей для розробки мобільних прикладних програм

З 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 вимагають більшої взаємодії з користувачем в порівнянні з десктопними і веб-програмами, як правило, очікуючи дій з боку користувача (наприклад, натискання кнопки), перш ніж відповідати користувачеві, оновленим інтерфейсом із зазначенням інформації, яка запитується. У розділі розглядається модель програмного забезпечення, яка фокусується на інтерактивних аспектах мобільних прикладних програм. Також проводиться аналіз загальної концепції розробки мобільних прикладних програм.

Модель Model View Controller (MVC) є популярним підходом для розробки мобільних прикладних програм. Зауважимо, 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 користувача. Таким чином, має сенс пов’язати компоненти інтерфейсу користувача з компонентами сховища даних. Однак через те, що компоненти інтерфейсу користувача оновлюються частіше, щоб задовольняти змінні вимоги користувачів та новіших технологій, ніж компоненти сховища даних, необхідно ввести додаткові компоненти. Метою такої схеми є поділ 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 – являє собою єдиний клас інтерфейсу користувача.

2. Сервіс – пов’язаний із завданнями, які будуть виконуватися у фоновому режимі потоків (наприклад, мережевих операцій), не зачіпаючи при цьому компоненти інтерфейсу користувача.

3. Постачальник контенту – дозволяє зберігати дані в 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 включає такий набір компонентів:

– компоненти інтерфейсу користувача;

– сервісні компоненти (місце розташування пристрою для відслідкування 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, обробки кліків користувачів і т. д.).

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

Вони також обробляють запити від `View` і `Controller` для отримання інформації про їхній стан та інструкції щодо зміни їх стану. Компоненти віджета аналогічні компонентам `Controller` у тому, що вони очікують введення з боку користувача (наприклад, натискання кнопки), а потім інтерпретують ці введення 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, і кожен компонент програми має своє відображення в `Model`, `View` і `Controller`. Однак при найближчому розгляді деталей все не так просто. Прикладом може служити компонент `Activities`, який є складовою частиною програми. Кожна активність складається з Java-файлу і відповідного файлу макета XML. Розробник визначає розташування UI активності в файлі XML та реалізує функціональність і поведінку у файлі Java (поділ користувача інтерфейсу з логікою прикладної програми). Коли використовуються віджет-компоненти, необхідно спочатку визначити віджети як елементи XML-файлу макета для створення екземпляра віджета і його атрибутів. Обробка поведінки віджета у відповідь на дії користувача також реалізуються у файлі Java. Проте розробник має доступ до атрибутів віджета з макета XML при реалізації його функціональних можливостей, і це порушує принцип поділу View з `Model/Controller` шаблону MVC.

Можливе рішення цієї проблеми полягає у використанні шаблону

Observer (Спостерігач) в MVC (рис. 1.4). Можна реалізувати інтерфейс Observer в компоненті Активності (View), щоб отримувати 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 поданням.

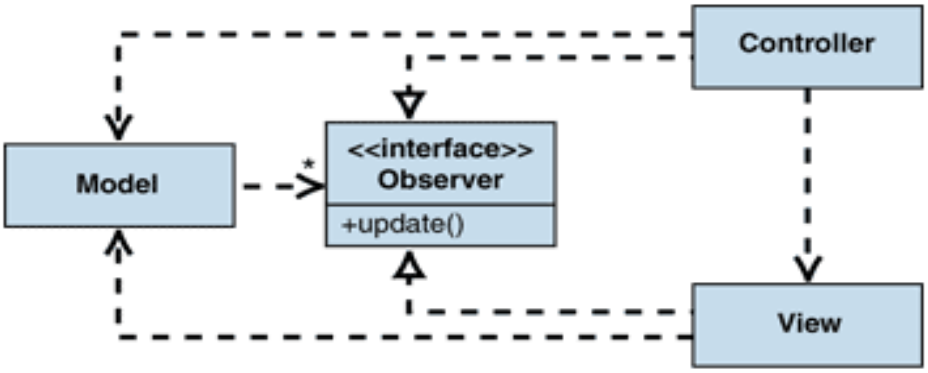

Рисунок 1.4 – Використання Observer у рамках MVC

Шаблон Layered Abstraction складається з ієрархії рівнів. На кожному рівні, ми маємо компоненти, які працюють разом в межах того ж рівня абстракції, і пов'язані з рівнем нижче для забезпечення функціональних можливостей вищого рівня.

Архітектура системи Android дотримується шаблону Layered Abstraction, як показано на рис. 1.5. Вона складається з чотирьох шарів абстракції з прямим зв'язком тільки між поточним шаром та 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 системи Android, а також реляційної системи управління базами даних – SQLite.

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

З точки зору розробника прикладних програм з великою кількістю компонентів для інтерфейсу користувача було б доцільно використовувати шаблон MVC. Використання шаблону дає можливість зменшити кількість програмного коду, зберігаючи при цьому ядро бізнес-логіки прикладної програми, при зміні вимог користувача.

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

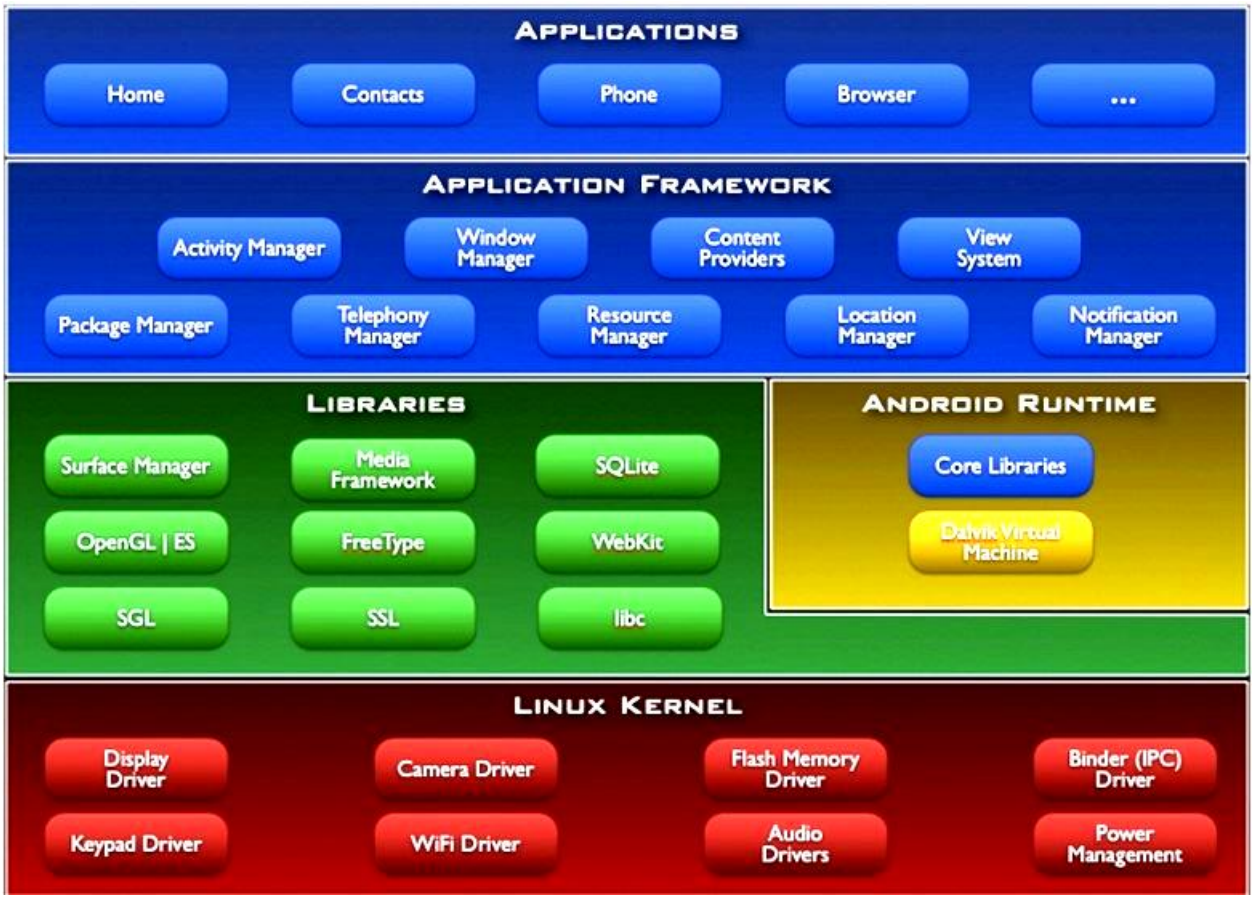

Рисунок 1.5 – Архітектура системи Android

1.7. Огляд засобів розробки мобільних прикладних програм

Комплект розробки програмного забезпечення Android (SDK) включає в себе повний набір інструментів розробника. [5] Він містить налагоджувач, бібліотеки, емулятор мобільного пристрою, оснований на QEMU, документацію, зразки коду та підручники (табл. 1.1).

Таблиця 1.1 – SDK Android

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

В даний час підтримуються платформи розробки, які працюють на операційній системі Linux (будь-який сучасний робочий стіл Linux), Mac OS X 10.5.8 або пізнішій версії та Windows XP або пізнішій версії. Станом на березень 2015 року SDK не доступна на Android, але розробка програмного забезпечення можлива за допомогою спеціалізованих програм для Android [6 – 8].

Приблизно до кінця 2014 року інтегрованим середовищем розробки (IDE), що офіційно підтримувалось, було Eclipse (рис. 1.6), яке пропонувало для розробки мобільних прикладних програм спеціальний плагін – інструменти розробки Android (ADT), хоча середовище IntelliJ IDEA (всі випуски) повністю підтримує Android-розробку «з ящика» [9] (рис. 1.7). NetBeans IDE також підтримує Android-розробку за допомогою плагіна [10] (рис. 1.8).

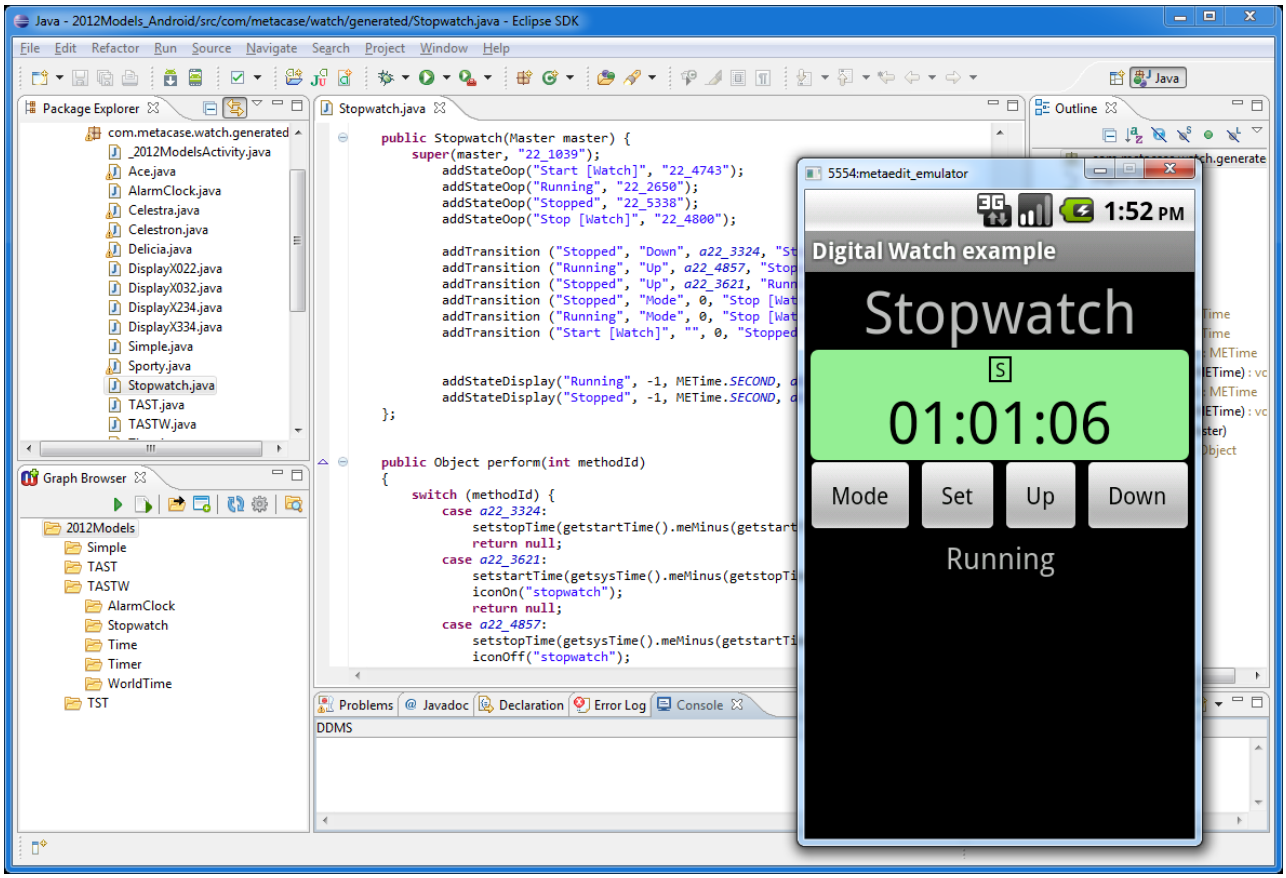

Рисунок 1.6 – IDE Eclipse

У 2015 році Google спільно з IntelliJ представили офіційну IDE від Google Android Studio [11]. Однак розробники можуть використовувати інші інтегровані середовища розробки. Крім того, розробники можуть використовувати будь-який текстовий редактор для редагування XML та Java-файлів, а потім використовувати інструменти командного рядка (комплект розробки Java і Apache Ant), щоб створити, скомпілювати та налагодити прикладні програми Android, а також керувати підключеними пристроями Android (наприклад, викликавши

перезавантаження, установлення програмного забезпечення, видалення пакетів) [12].

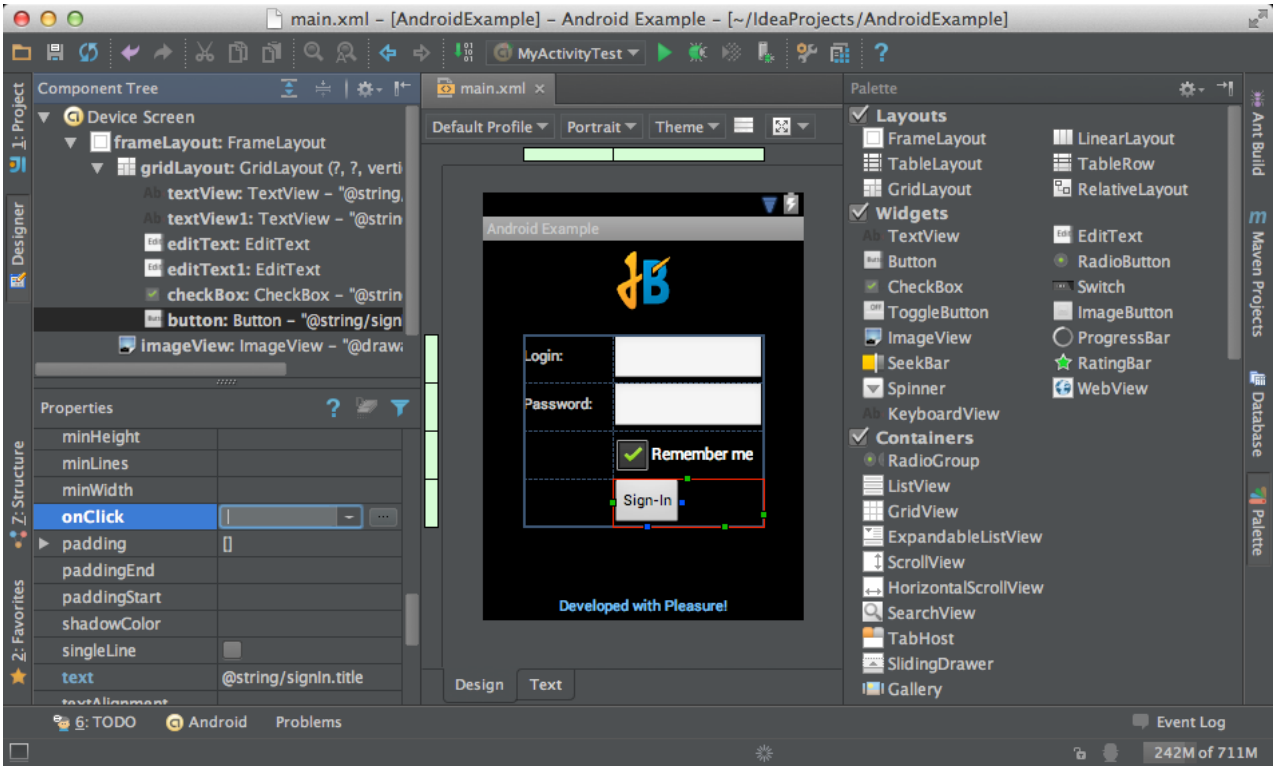

Рисунок 1.7 – IntelliJ IDEA

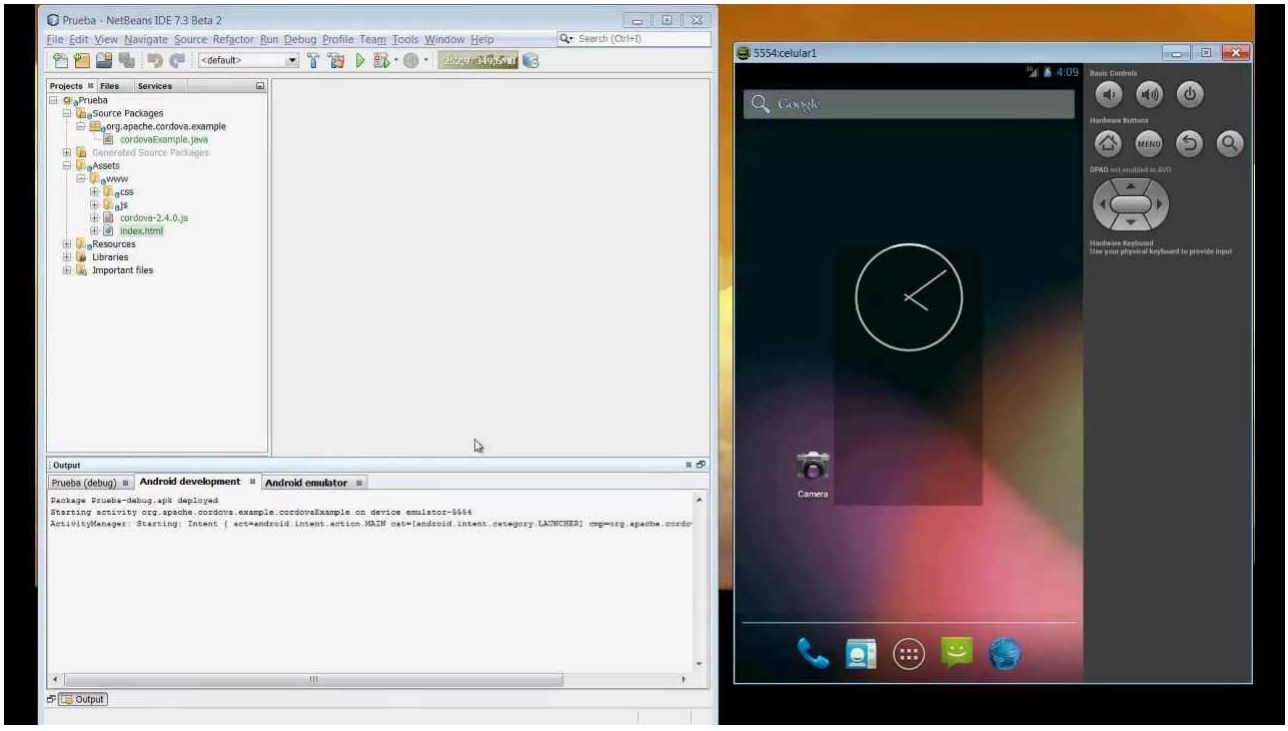

Рисунок 1.8 – NetBeans IDE

Покращення в SDK Android йдуть пліч-о-пліч з загальним розвитком Android-платформи. SDK також підтримує старі версії Android-платформи, якщо розробники хочуть запускати свої прикладні програми на старих пристроях.

Засоби розробки є компонентами, що завантажуються, таким чином, що після завантаження останньої версії та платформ, старі платформи та інструменти розробки також можуть бути завантажені для тестування сумісності [13].

Додатки Android упаковуються у файли у форматі `.apk` та зберігаються в папці `/data/app` на пристрої Android (папка доступна тільки для суперкористувача з міркувань безпеки). Пакети `apk` містять файли `.dex` [14] (байт-код, що виконується прикладною програмою для Dalvik VM), файли ресурсів та ін.

ADB (Android Debug Bridge). Це – клієнт-серверна прикладна програма, яка надає доступ до працюючого емулятора або пристрою (рис. 1.9). З її допомогою можна копіювати файли, встановлювати 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`, а тепер вона знаходиться в папці `platform-tools`.

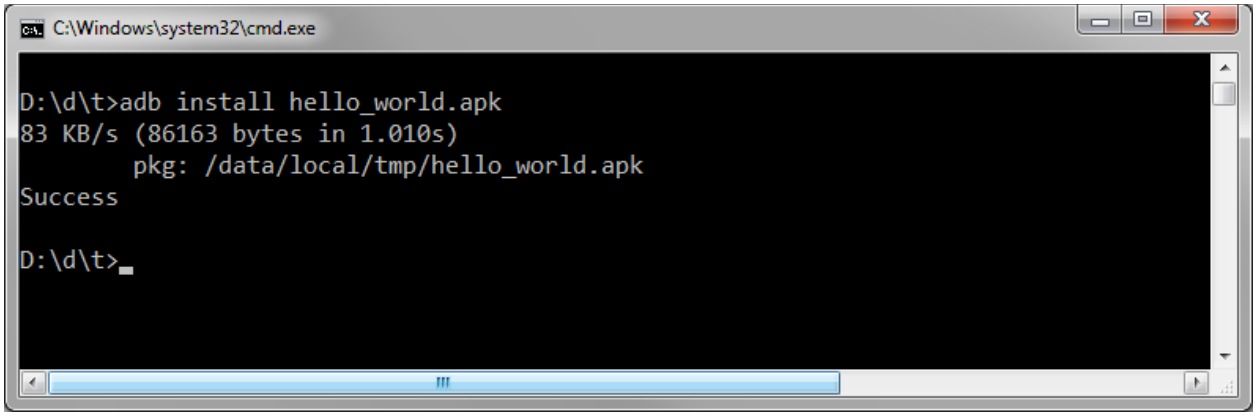

Рисунок 1.9 – Установлення пакета `apk` з використанням `adb`

ADB складається з трьох компонентів: фонової служби (демона), що працює в емуляторі; сервісу, запущеного на комп'ютері розробника; клієнтської програми (на зразок DDMS), яка зв'язується зі службою через Сервіс.

Fastboot. Fastboot є діагностичним протоколом, який йде в комплекті з SDK та використовується в першу чергу для зміни флеш файлової системи через USB-з'єднання з комп'ютером. `Fastboot` вимагає, щоб пристрій запускався завантажувачем в режимі `Second Program Loader`, в якому виконуються тільки найосновніші ініціалізації обладнання. Після включення протоколу на самому пристрої, він буде приймати певний набір команд, які надіслані до нього через USB, використовуючи командний рядок. Деякі з найбільш часто використовуваних команд швидкого завантаження:

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

- erase – видаляє певний розділ;
- reboot – перезавантажує пристрій або в основну операційну систему, або назад в завантажувач;
- devices – відображає список всіх пристроїв (з серійним номером), підключених до комп'ютера;
- format – форматує певний розділ; файлова система розділу повинна розпізнаватися пристроєм.

Android NDK. Android Native Development Kit наведено в табл. 1.2. Бібліотеки, написані на C, C ++ та інших мовах можуть бути скомпільовані в ARM, MIPS або в машинному коді x86 та встановлені на пристрій з використанням набору Android NDK. Рідні класи можна викликати з Java-коду, що виконується під Dalvik VM, використовуючи виклик `System.loadLibrary`, який є частиною стандартних класів Java в Android. [15, 16].

Таблиця 1.2 – 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-платформна
Доступна	Англійська
Тип	IDE, SDK
Сайт	developer.android.com/tools/sdk/ndk/index.html

Розроблені прикладні програми можуть бути скомпільовані та встановлені за допомогою традиційних інструментів розробки [17]. Тим не менш, відповідно до Android-документації, NDK не повинен використовуватися виключно для розробки прикладних програм тільки тому, що розробник вважає за краще програмувати на C/C++, тому що використання NDK збільшує складність прикладної програми, що не піде йому на користь.

ADB-налагоджувач дає права `root` в `Android Emulator`, що дозволяє завантажувати та виконувати програмний код, оптимізований під ARM, MIPS або x86-процесори. Машинний код може бути скомпільований з використанням GCC або Intel C ++ Compiler на стандартному ПК. Запуск машинного коду на Android-платформі ускладнюється використанням нестандартної бібліотеки C (Libc, відомої як Bionic). Графічна бібліотека Android, яка використовується для арбітражу та контролю доступу до цього пристрою, називається Graphics Library Skia (SGL), вона випущена 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 [18].

На відміну від додатків Java, основаних на використанні IDE, таких, як Eclipse, NDK оснований на командному рядку та вимагає введення команд вручну 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 (рис. 1.10). ADK – це Arduino-сумісна платформа, що підключається до Android-пристрою через USB або Bluetooth та містить безліч датчиків, сенсорів та індикаторів.

Рисунок 1.10 – ADK 2012

Google пропонує два напрямки застосування ADK:

1) комерційний – аудіодок-станції, інтеграція в спортивні тренажери та ін.;

2) хобі – контролери роботизованої техніки.

ADK 2012 містить:

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

2. Два USB. Перший – для підключення до Android-пристрою, другий – для налагодження та програмування.

3. Датчики світла, кольору, наближення, температури, вологості, атмосферного тиску та акселерометр.

4. Слот для SD-карти.

5. Підтримка Bluetooth.

6. Шість семисегментних світлодіодних RGB-матриць, 12 «party mode» світлодіодів.

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

8. Підсилювач звуку та динамік.

9. NFC-мітка, доступна для запису.

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

Сам комплект ADK 2012 має вигляд закінченого пристрою у формі будильника з функцією аудіодока.

Вбудована підтримка Go. Починаючи з версії 2.4, для програмістів, які пишуть Android-програми мовою програмування Go, включена підтримка мови без будь-якого Java-коду, хоча і з обмеженим набором інтерфейсів Android.

Android APIMiner. Це платформа, яка є інструментом автоматичної генерації та вилучення документації Javadoc з реальних прикладних програм Android з відкритим вихідним кодом та з прикладами використання. Для поліпшення якості витягнутих прикладів APIMiner використовує внутрішньопроцесуальний статичний алгоритм витягування [19] .

У табл. 1.3 подано опис основних компонентів Android APIMiner, а на рис. 1.11 відображено процес взаємодії між ними.

Таблиця 1.3 – Компоненти Android APIMiner

Компонент	Опис
Source Code Analyzer	Цей модуль аналізує вихідний код API та кліентських систем для «витягування» структурних даних
Patterns Analyzer	Цей модуль витягує шаблони використання елементів API на основі їх використання
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

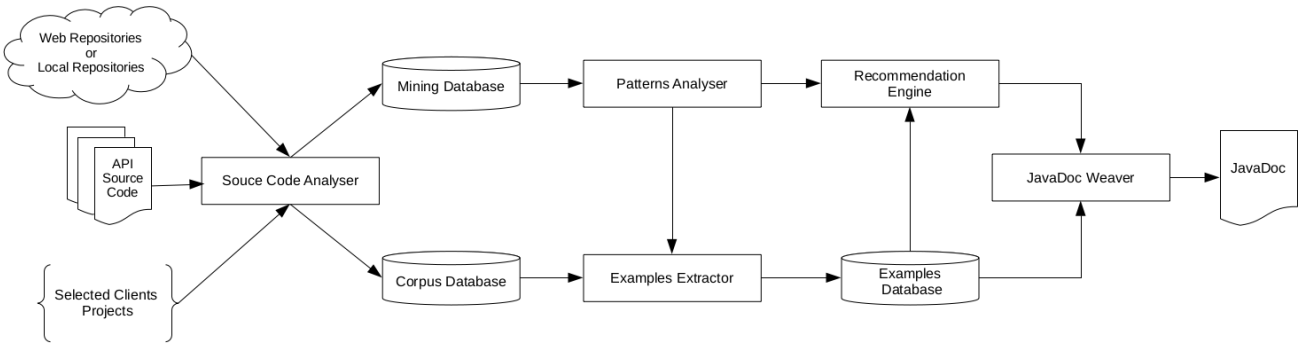

Рисунок 1.11 – Компоненти Android APIMiner

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

TK-інструментарій для створення графічного інтерфейсу користувача піднімає розробку настільних прикладних програм на більш високий рівень, ніж звичайні засоби. TK є стандартом інтерфейсу не тільки для TCL, але і 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 [20].

Деякі особливості AndroWish:

1. Рідний TCL/TK 8.6 порт для платформи Android (починаючи з версії 3.3.3 або вище) для процесорів ARM і x86.

2. Виконання існуючих 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 з використанням движка шрифтів Freetype.

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

8. Включає в себе віджет `tkpath`, який розширює можливості канви SVG, такі, як рендеринг, альфа-канали, підтримка `TrueType` контурних шрифтів.

9. Деякі конкретні об’єкти Android доступні через SDL та викликаються командою `sdltk`.

10. Використання AndroWish вимагає Android SDK та Android NDK.

11. Тестування та налагодження сценаріїв TCL на Android-пристроях може здійснюватися з системи розробки з використанням tkconclient. Файли можуть бути передані за допомогою підключення SSH/SFTP, як описано в `tkconclient`.

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

App Inventor для Android. 12 липня 2010 року Google оголосила про доступність App Inventor для Android (рис. 1.12). App Inventor – це веб-орієнтоване візуальне середовище розробки для програмістів-початківців, основане 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 в текст, контактів, даних постійного зберігання та веб-служб.

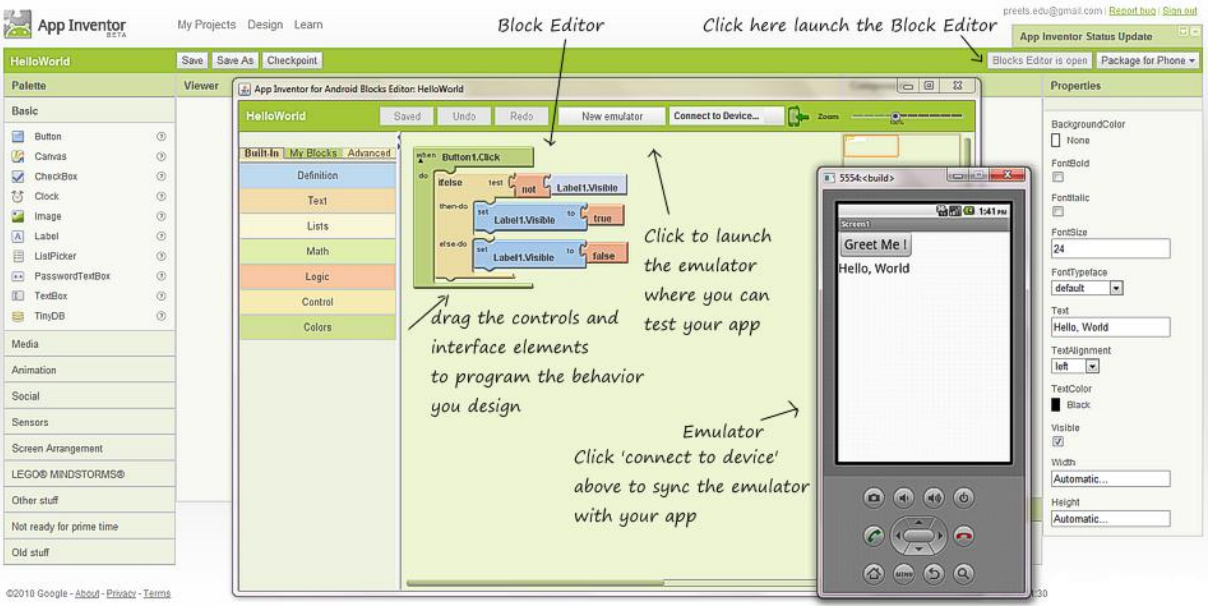

Рисунок 1.12 – App Inventor

Остання версія створена в результаті співробітництва Google та MIT, випущена у лютому 2012 року, в той час як перша версія, створена виключно MIT, була запущена у березні 2012 року та оновлена до App Inventor 2 у грудні 2013 року. З 2014 року App Inventor підтримується виключно MIT [21].

Basic4android – VisualBasic Native Android Language. Це простий і потужний інструмент розробки прикладних програм для пристроїв, що працюють під управлінням операційної системи Android. Мова Basic4Android (рис. 1.13) дуже схожа на популярну мову Visual Basic. При розробці прикладних програм використовується безліч різних додаткових бібліотек. Для виконання створених програм ніяких додаткових runtime-засобів не потрібно.

Corona SDK. Є комплектом розробки програмного забезпечення, створеним Вальтером Лухом, засновником Corona Labs Inc. Він дозволяє програмістам створювати мобільні прикладні програми для iPhone, IPAD та Android-пристроїв (рис. 1.14).

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

Середовище Delphi також може бути використане для створення прикладних програм для платформи Android із застосуванням мови Object Pascal.

Остання версія Delphi XE8 була розроблена Embarcadero Studio.

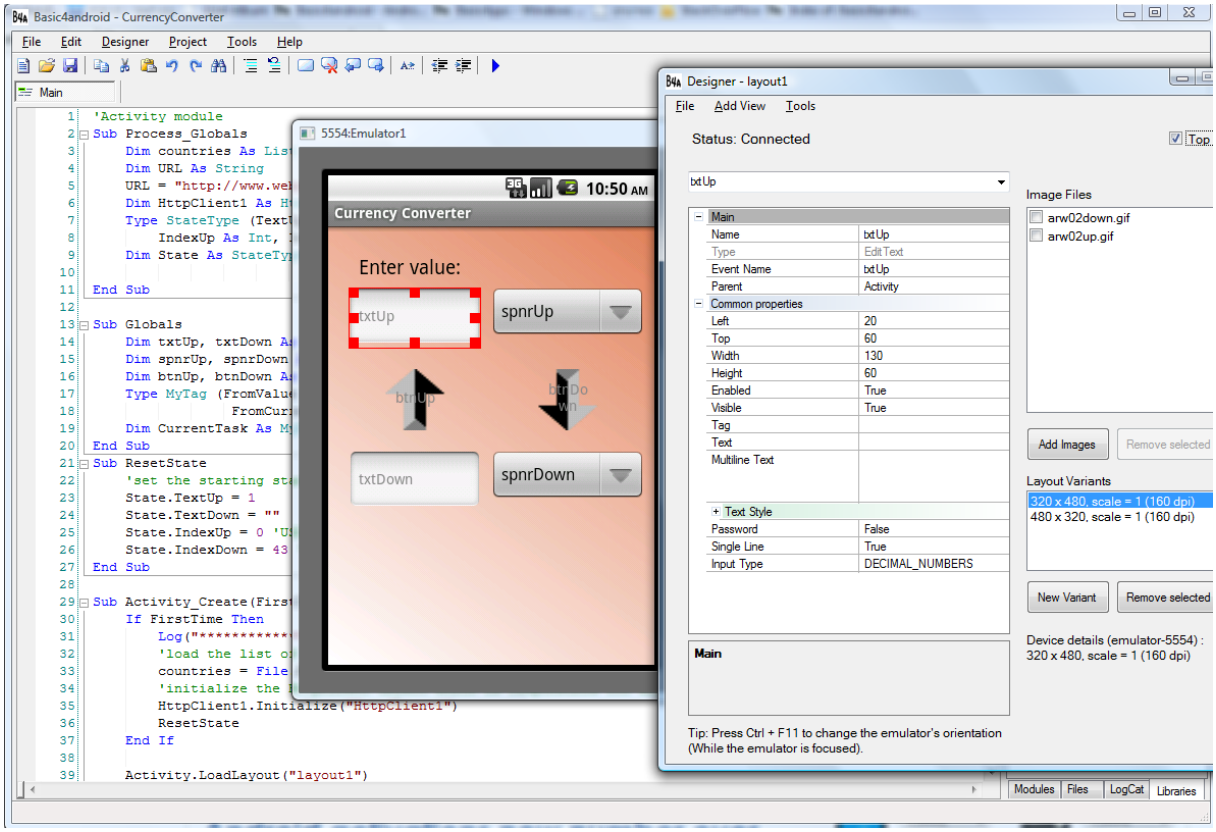

Рисунок 1.13 – Basic4android

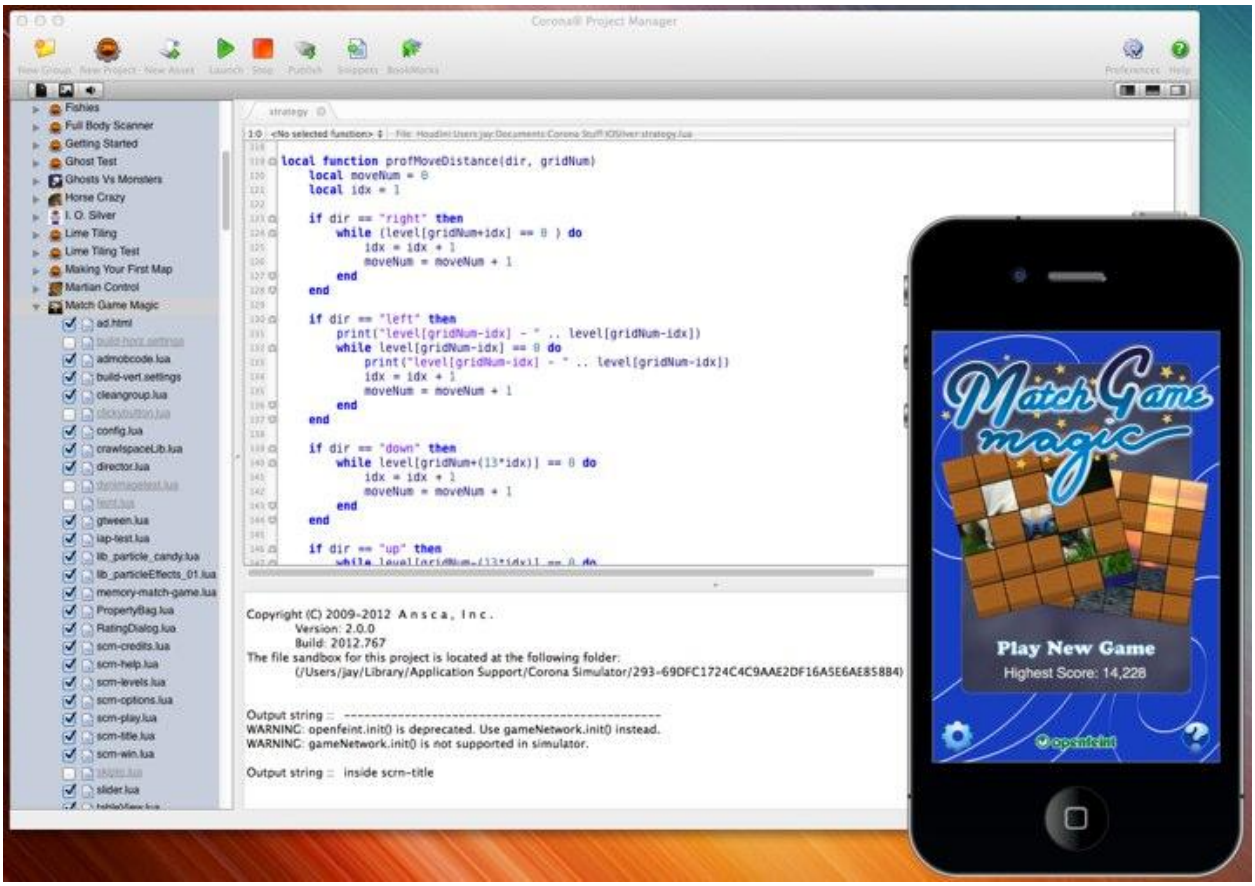

Рисунок 1.14 – Corona SDK

*Embarcadero® RAD Studio XE*8 (рис. 1.15) є закінченим рішенням для розробки програмного забезпечення для Windows, Mac, IOS, Android та IoT. RAD Studio дозволяє будувати готові рішення, які розробляються не тільки для клієнтських платформ, але також і для мобільних пристроїв, смарт-пристроїв, таких, як смарт-годинник та інші гаджети IoT [22].

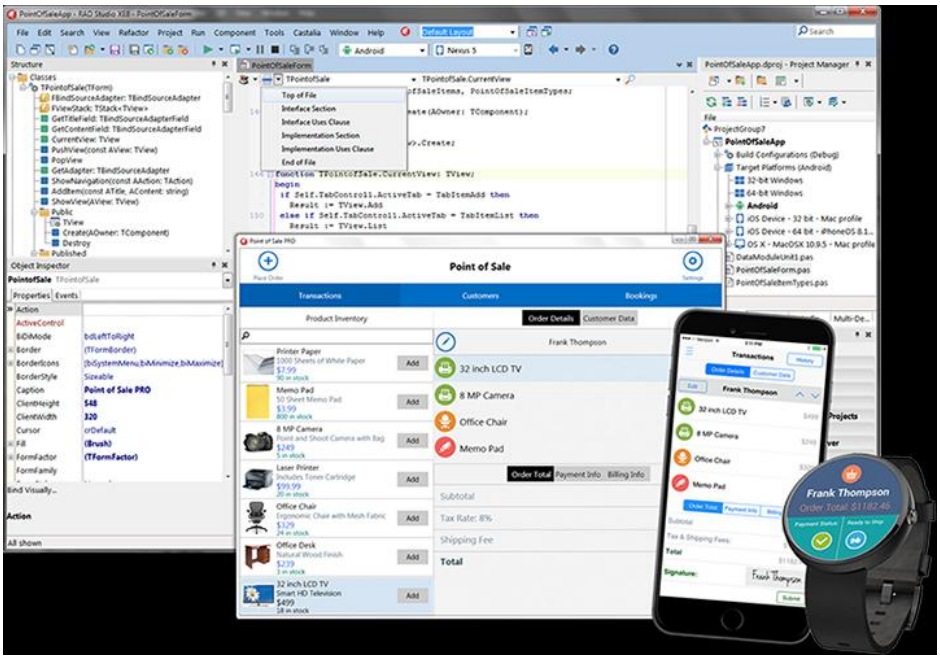

Рисунок 1.15 – RAD Studio XE8

HyperNext Android Creator (HAC) є системою розробки програмного забезпечення, орієнтованою на програмістів-початківців [23], яка допомагає їм створювати свої власні прикладні програми для Android, не знаючи Java та Android SDK (рис. 1.16). HAC основана на тому, що HyperCard обробляє програмне забезпечення у вигляді стопки карток, при цьому в будь-який момент часу видно тільки одну картку, що дуже добре підходить для прикладних програм мобільних телефонів, в яких відображається одноразово тільки одне вікно.

Kivy – це відкрита бібліотека мовою Python для розробки прикладного програмного забезпечення «мультитач» з природним інтерфейсом користувача (NUI) для широкого набору пристроїв (рис. 1.17). Kivy забезпечує можливість підтримки єдиної прикладної програми для численних операційних систем («кодуй один раз, запускай скрізь»). Kivy має виготовлений на замовлення інструмент розгортання мобільних прикладних програм під назвою Buildozer, доступний тільки для Linux. Buildozer в даний час існує в альфа-версії, але він набагато зручніший, ніж громіздкі старі методи розгортання Kivy. Прикладні програми, запрограмовані Kivy, можуть бути подані на будь-якій мобільній платформі Android-прикладних програм.

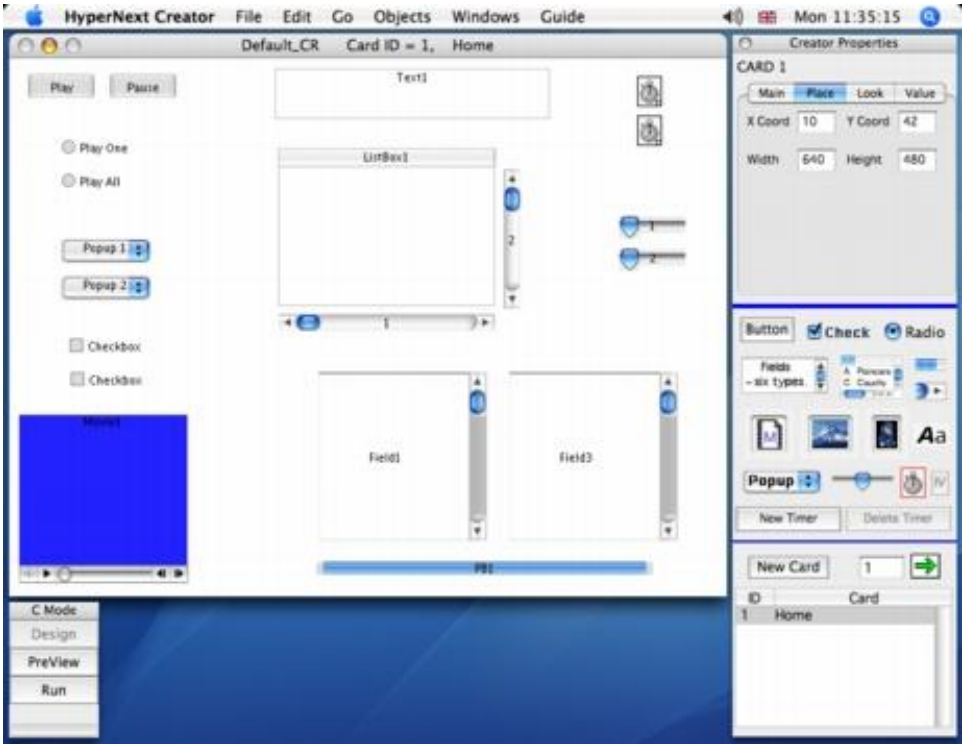

Рисунок 1.16 – HyperNext Android Creator

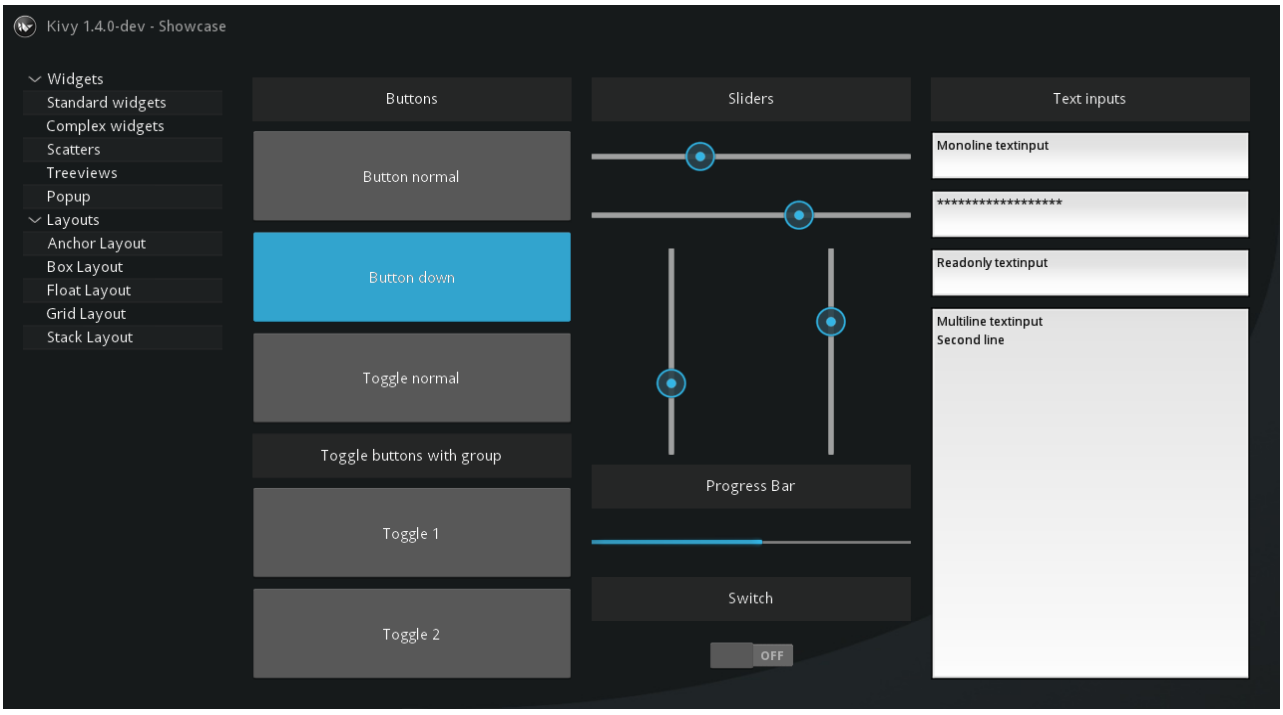

Рисунок 1.17 – Kivy 2.4 on Android

Lazarus можна використовувати для розробки прикладних Android-програм (рис. 1.18) на мові Pascal з компілятором Free Pascal, починаючи з версії 3.7.2.

Qt для Android, починаючи з версії Qt 5, створює прикладні програми для запуску на пристроях з Android v3.3.3 (рівень API 10) або пізнішої версії. Qt є основою для крос-платформних прикладних програм, які можуть запускатися

на цільових платформах, таких, як Android, Linux, IOS, Sailfish OS і Windows. Розробку Qt-прикладних програм виконують з використанням мови C++ та QML, вимагаючи при цьому встановлених Android NDK та SDK. Qt Creator є інтегрованим середовищем розробки і спільно з Qt Framework використовується для розробки мультиплатформних прикладних програм.

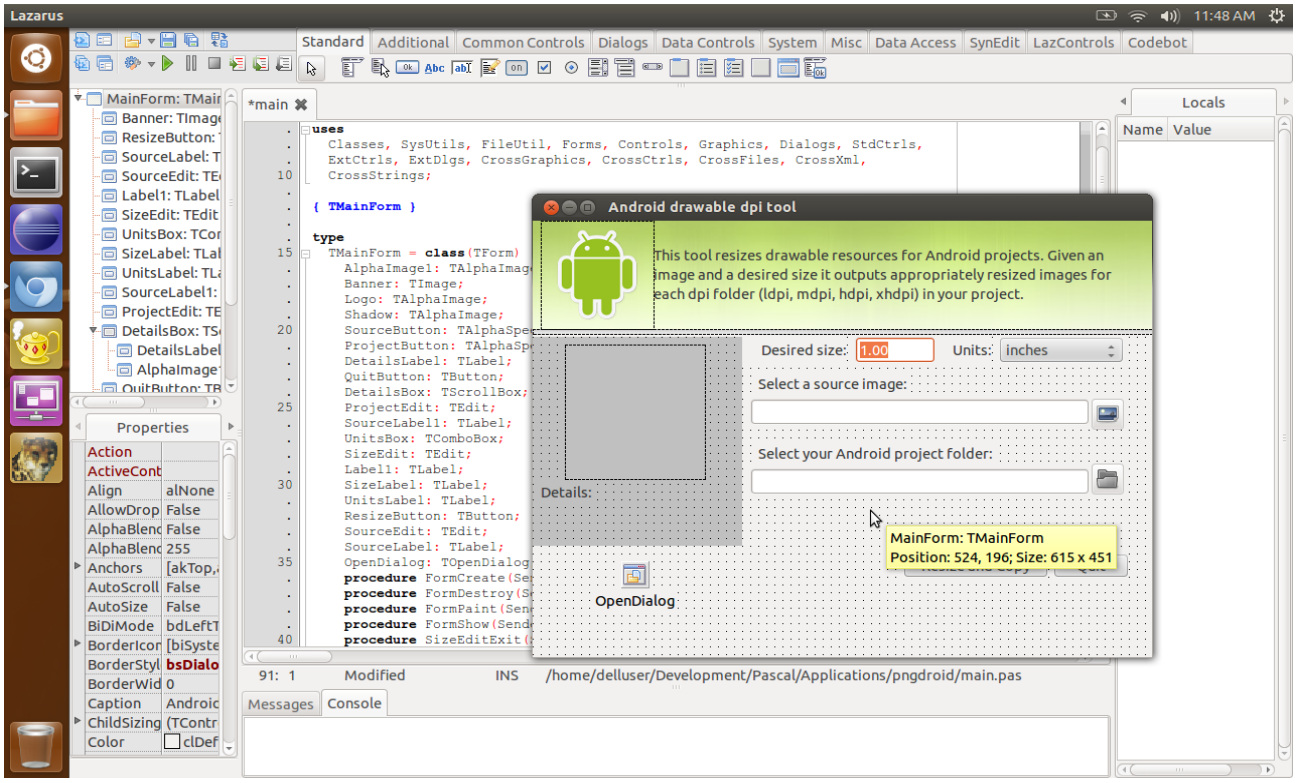

Рисунок 1.18 – Проект Lazarus

RFO BASIC – це діалект Dartmouth Basic, що є інтерпретатором з набором бібліотек для доступу до апаратного обладнання, датчиків, звуку, графіки, мультитачу, файлової системи, SQLite, мережі, HTML-інтерфейсу, шифрування, SMS, функцій телефону, електронної пошти, перетворення тексту в 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 року.

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 в цілому набором Java API від Ruby, при цьому можливе використання сторонніх бібліотек Java.

Saphir є відгалуженням з відкритим вихідним кодом від проекту Rebol3 (R3). Вся функціональність R3, в тому числі GUI, графіка, доступ до мережі, доступ до файлів, парсинг та інші особливості портується на основні пор-

тативні 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) доповнюється простим утилітарним дизайном Saphir.

Бібліотека SDL пропонує, крім можливості розробки з використанням Java, можливість розробки з використанням C з наступним простим перенесенням існуючих SDL та власних прикладних програм C. Застосування Java-ін'єкцій та прокладок JNI дозволяє використовувати рідну бібліотеку SDL при портуванні на пристрої Android, наприклад, як у відео грі Jagged Alliance 3.

Метою The Simple project є забезпечення розробника простою в розумінні та використанні мовою для розробки прикладних програм для платформи Android. [24] The Simple project є основним діалектом для розробки прикладних програм Android. Він націлений 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) та 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, [25] останнє оновлення вихідний код зазнавав у серпні 2009 року.

Visual Studio 2015 (рис. 1.19) підтримує розробку крос-платформних прикладних програм, дозволяючи розробникам C++ створювати проекти з шаблонів для Android-прикладних програм або динамічні високопродуктивні колективні бібліотеки для включення їх в інші рішення. Функціонал середовища включає в себе інтелектуальний підказувач IntelliSense, точки зупину, розгортання пристроїв та емуляції. [26]

WinDev Mobile – власна IDE, яка створена PC SOFT (рис. 1.20), що використовується для створення графічного інтерфейсу користувача (GUI) прикладних програм для смартфонів та планшетів (включаючи Android-пристрої). Вона використовує мову програмування WLanguage та доступна англійською, французькою та китайською мовами.

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

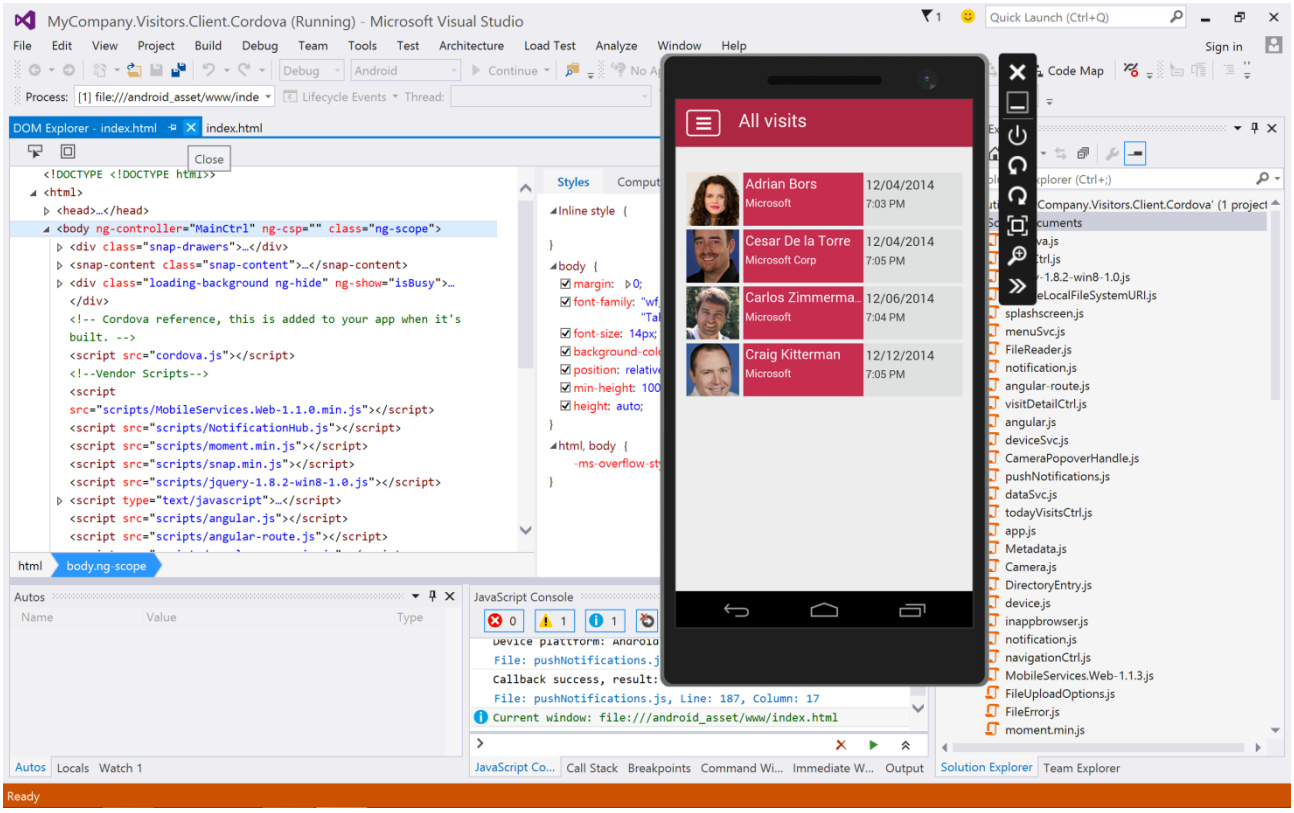

Рисунок 1.19 – Visual Studio 2015

Рисунок 1.20 – WinDev Mobile

X11 Basic є діалектом мови програмування Basic з графічними можливостями, який об'єднує такі функції, як оболонки сценаріїв, програмування CGI та повна графічна візуалізація. Синтаксис в основному схожий на старий GFA Basic, який використовувався на комп'ютерах Atari ST.

1.8. Огляд інтегрованих середовищ розробки (IDE)

Інтегроване середовище розробки (IDE) є інструментом, який дозволяє розробнику прикладних програм виконати повний цикл розробки програмного забезпечення. Цей цикл включає проектування, кодування, компіляцію, тестування, налагодження та упаковку програмного забезпечення прикладної програми. Для створення Android-додатків Google в даний час підтримує дві IDE:

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

– Android Studio http://developer.android.com/sdk/installing/studio.html

Обидва середовища розробки потребують використання мови Java.

Середовища розробки від Google та мова програмування Java не є єдиними варіантами для розробки 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). Така розробка відома як крос-платформна розробка. Існує багато альтернатив інструментам Google. Огляд альтернативних IDE наведено в табл. 1.4. У таких системах код може бути написаний різними мовами, такими, як BASIC, HTML5 або Lua. Багато з альтернативних IDE можуть використовуватися вільно, деякі мають відкритий вихідний код, деякі є обмеженими версіями. Деякі вимагають Android Software Development Kit (SDK), який поставляється разом з інструментами Google. Можна встановити кілька середовищ розробки на одному комп'ютері, щоб протестувати їх можливості.

Таблиця 1.4 – Огляд альтернативних середовищ розробки

Найменування	**Мова програмування**	**Крос-платформність**	**URL**
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/

Продовження табл. 1.4

Найменування	Мова програму-вання	Крос-платформ-ність	URL
Intel XDK	HTML5	Так	https://software.intel.com/en-us/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/
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/

Запитання для самоконтролю

1. Назвіть рівні програмного забезпечення Android.
2. Назвіть засоби керування ресурсами мобільних пристроїв ОС Android.
3. Дайте характеристику віртуальній машині Dalvik.
4. Дайте характеристику віртуальній машині ART.
5. Назвіть обмеження на використання Java API у мобільному пристрої.
6. Наведіть основні засоби розробки мобільних додатків.
7. Наведіть основні середовища розробки IDE.

2. АРХІТЕКТУРА МОБІЛЬНИХ ДОДАТКІВ

2.1. Компоненти інтерфейсу

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

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

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

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

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

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

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

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

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

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

2.1.1. Створення XML

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

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

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

Після оголошення макета в файлі XML збережіть файл з розширен-

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

2.1.2. Завантаження ресурсу XML

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

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

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

2.1.3. Атрибути

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

2.1.4. Ідентифікатор

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

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

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

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

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

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

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

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

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

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

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

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

2.1.5. Параметри макета

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

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

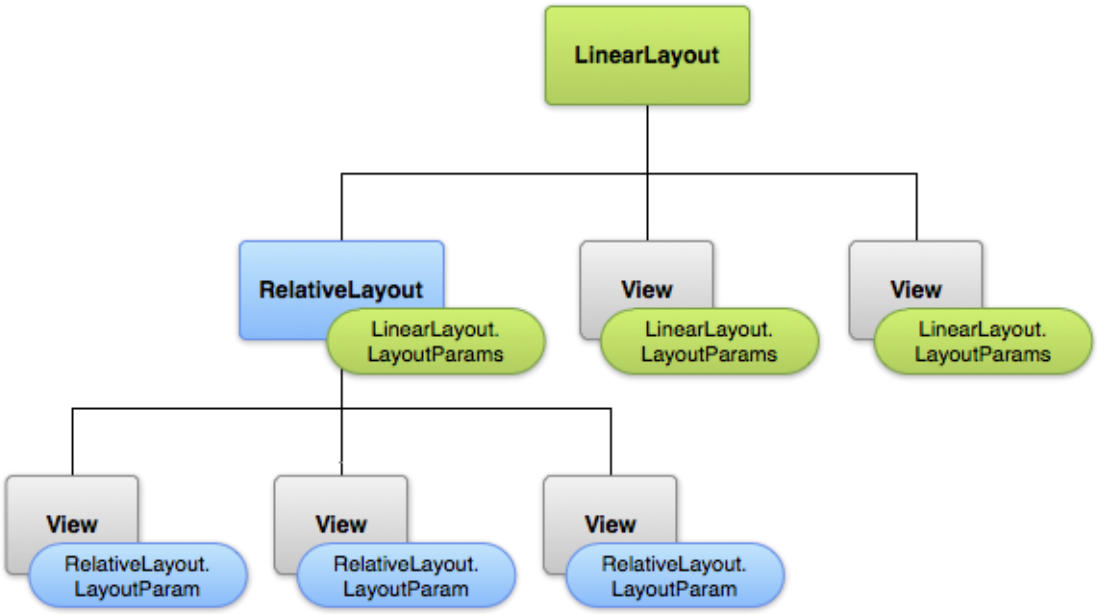

Рисунок 2.1 – Графічна інтерпретація ієрархії подання з параметрами макета кожного подання

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

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

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

– `wrap_content` – розмір подання задається за розмірами його вмісту;

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

подань.

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

2.1.6. Розміщення макета

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

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

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

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

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

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

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

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

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

2.1.7. Макети інтерфейсу користувача

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

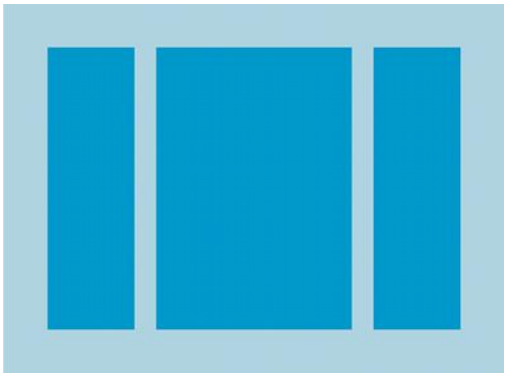

Рисунок 2.2 – Лінійний макет

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

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

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

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

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

Приклад

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:orientation="vertical" >
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/to" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/subject" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="@string/message" />
    <Button
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:layout_gravity="right"
        android:text="@string/send" />
</LinearLayout>
```

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

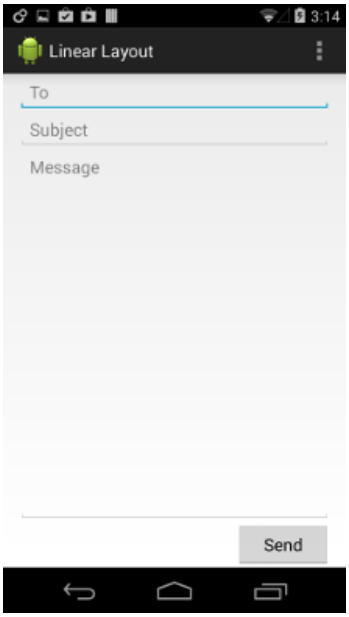

Рисунок 2.3 – Лінійний макет (приклад)

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

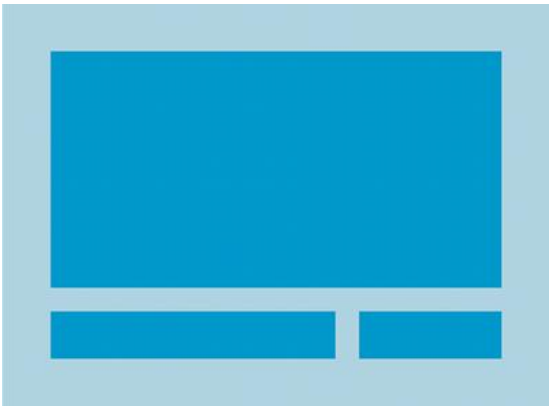

Рисунок 2.4 – Відносний макет

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

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

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

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

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

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

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

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

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

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

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

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

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

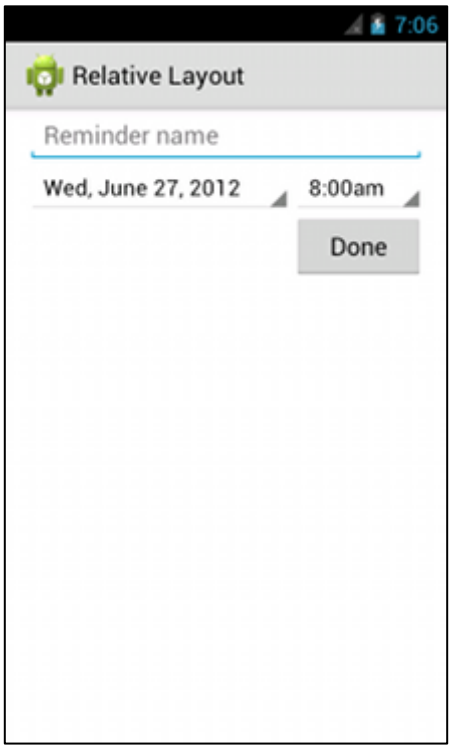

Рисунок 2.5 – Відносний макет (приклад)

Більш детальноїу інформацію про атрибути, доступні кожному поданню `RelativeLayout`, див. в `RelativeLayout.LayoutParams`.

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

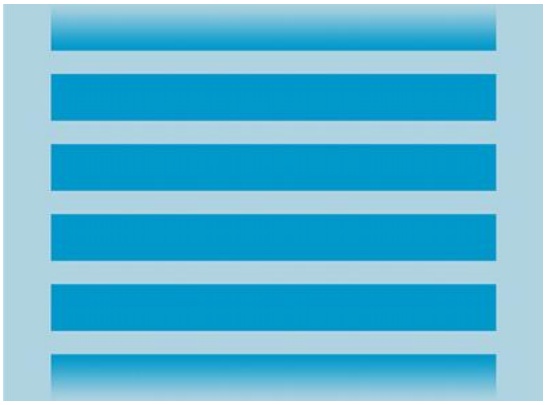

Рисунок 2.6 – Подання у вигляді списку

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    }

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

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

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

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

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

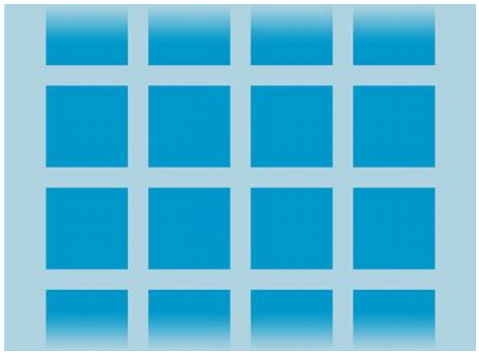

Рисунок 2.7 – Подання у вигляді сітки

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

Приклад. У цьому прикладі створюємо сітку ескізів зображень.

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

1. Створіть новий проект з назвою `HelloGridView`.

2. Знайдіть декілька фото, які б ви хотіли використовувати, або завантажте ці зразки зображень. Збережіть файли зображень у папку проекту `res/drawable/`.

3. Відкрийте файл `res/layout/main.xml` і вставте таке:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
        return imageView;
    }

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

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

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

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

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

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

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

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

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

6. Запустіть додаток.

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

2.2. Життєвий цикл візуальних компонентів

2.2.1. Операції (Activity)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Існує всього три стани операції:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Продовження табл. 2.1

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

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

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

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

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

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

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

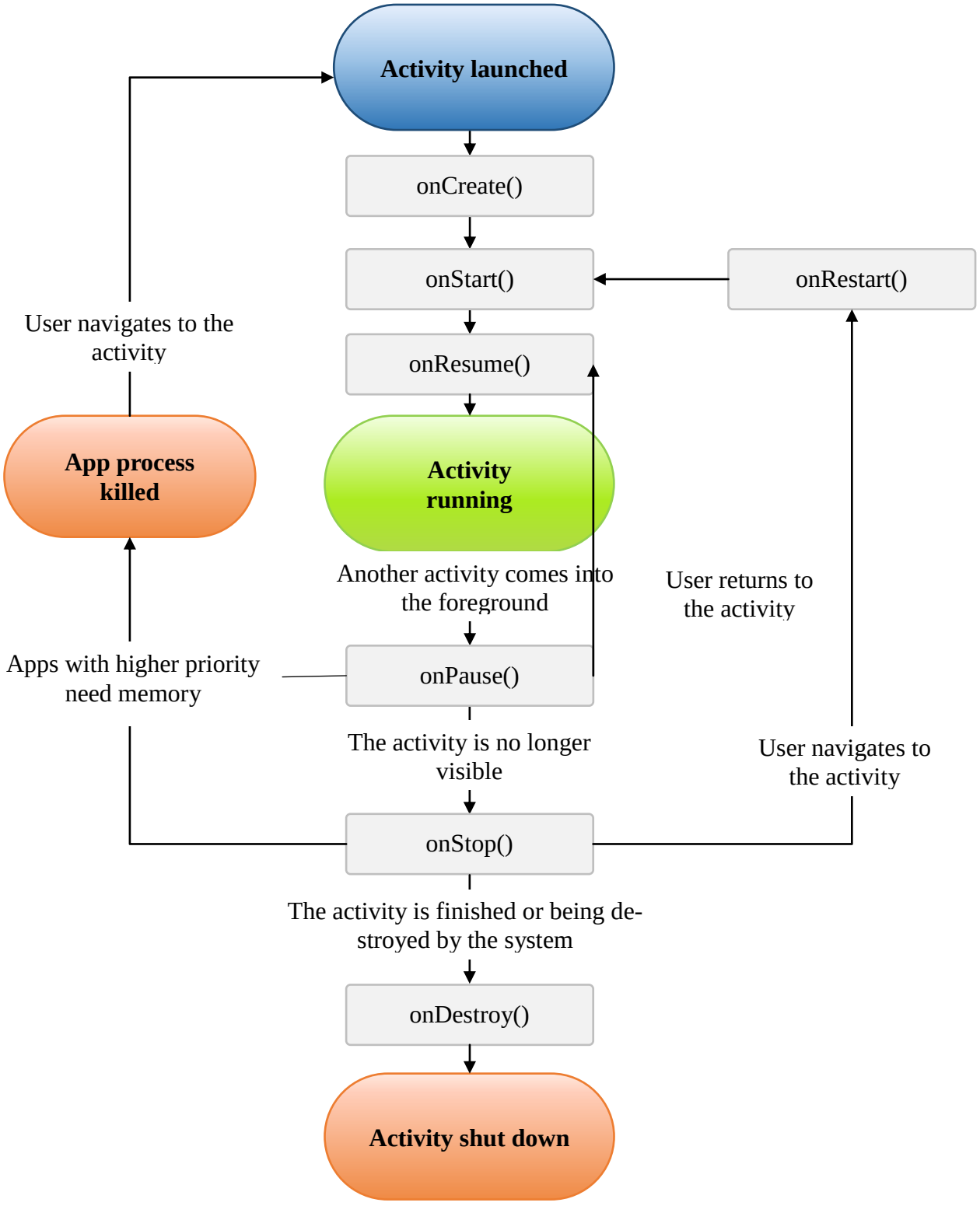

Рисунок 2.8 – Життєвий цикл операції

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

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

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

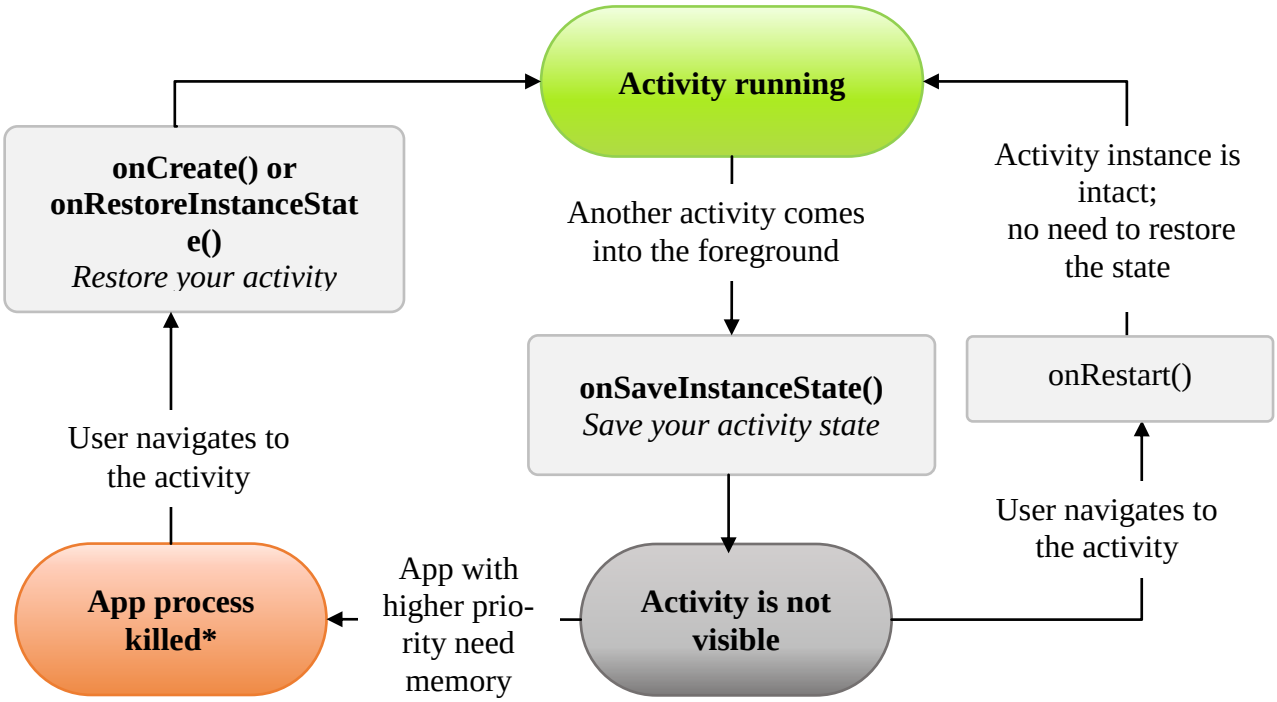

Рисунок 2.9 – Способи повернення операції до відображення для користувача в незміненому стані

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1. Виконується метод `onPause()` операції А. Послідовно виконуються методи `onCreate()`, `onStart()` та `onResume()` операції Б (тепер для користувача відображається операція Б).
2. Потім, якщо операція А більше не відображається на екрані, виконується її метод `onStop()`.

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

2.2.2. Фрагменти

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

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

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

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

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

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

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

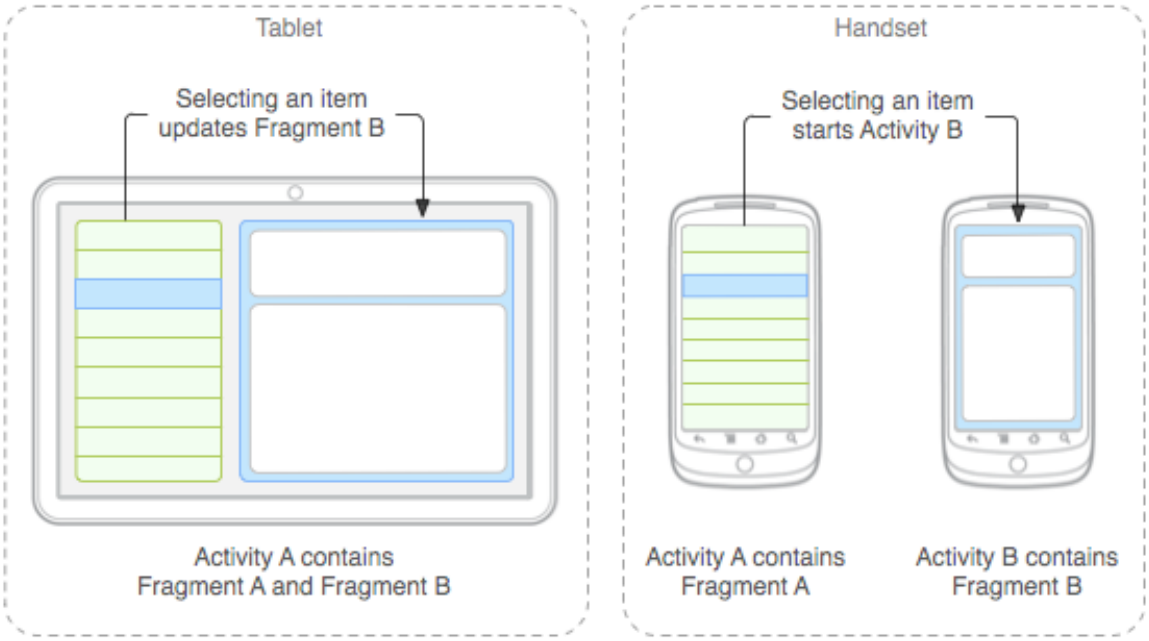

Рисунок 2.10 – Приклад об’єднання і перестановки компонентів

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

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

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

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

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

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

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

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

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

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

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

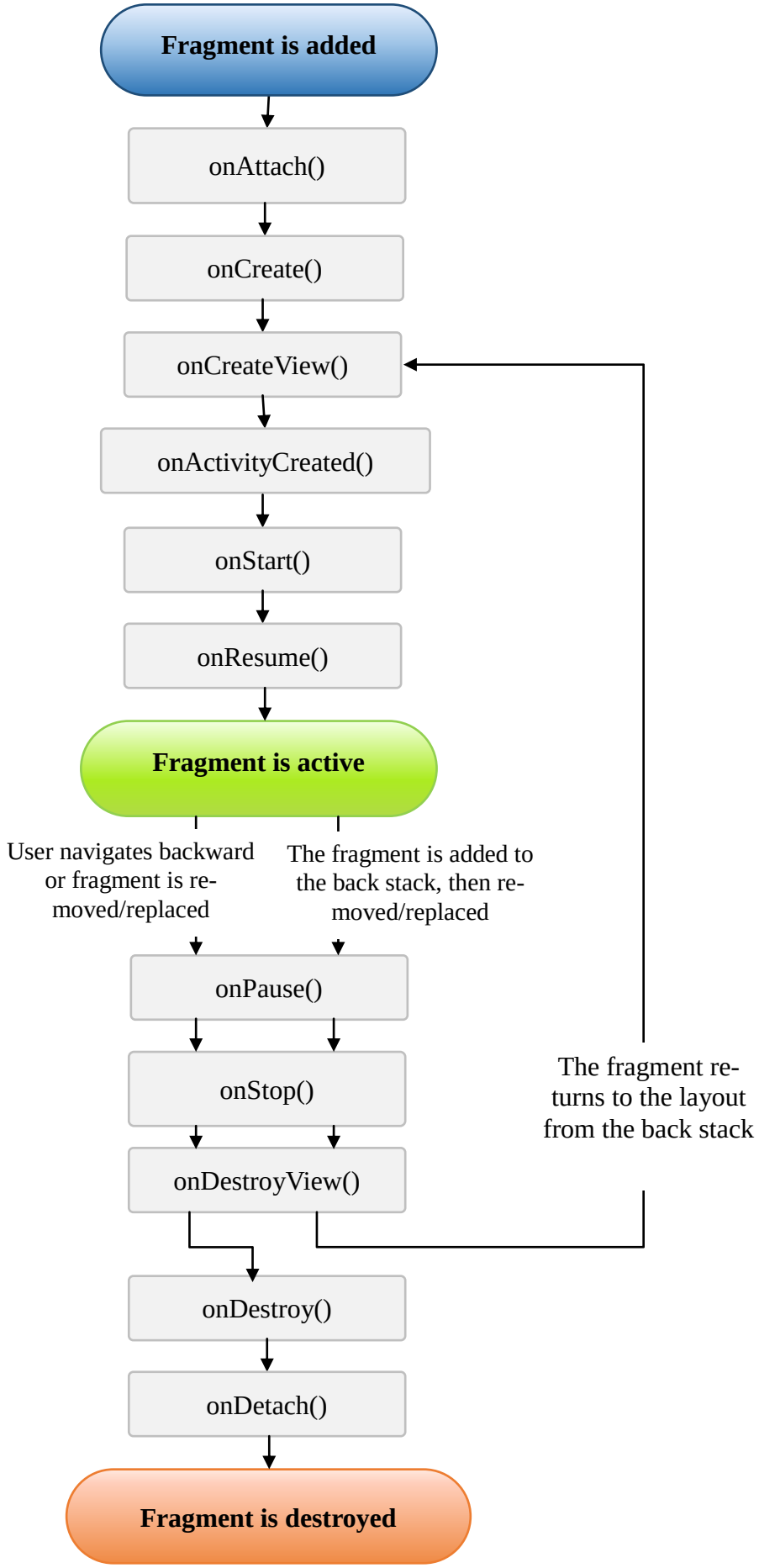

Рисунок 2.11 – Життєвий цикл фрагмента (під час виконання операції)

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

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

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

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

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

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

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

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

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

Метод `inflate()` приймає три аргументи:

– ідентифікатор ресурсу макета, розгортання якого слід виконати;

– об'єкт класу `ViewGroup`, який повинен стати батьківським для макета після розгортання. Передача параметра `container` необхідна для того, щоб система, до якої направляється макет, змогла застосувати параметри макета до кореневого подання розгорнутого макета, який визначається батьківським поданням;

– логічне значення, яке показує, чи слід прикріпити макет до об'єкта `ViewGroup` (другий параметр) під час розгортання (в даному випадку це *false*, тому що система вже вставляє розгорнутий макет в об'єкт `container`, і передача значення *true* створила б зайву групу подання в остаточному макеті).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Нижче вказані дії, які дозволяють виконати `FragmentManager`:

– отримувати фрагменти, наявні в операції, за допомогою методу `findFragmentById()` (для фрагментів, які надають інтерфейс користувача в макеті операції) чи `findFragmentByTag()` (як для фрагментів, що мають інтерфейс користувача, так і для фрагментів без нього);

– знімати фрагменти зі стека переходів назад методом `popBackStack()` (імітуючи натискання кнопки *Назад* користувачем);

– реєструвати процес-слухач змін в стеку переходів назад за допомогою методу `addOnBackStackChangedListener()`.

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

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

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

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

```
FragmentManager fragmentManager = getFragmentManager();

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

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

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

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

```
// Створити новий фрагмент і транзакцію

Fragment newFragment = new ExampleFragment();

FragmentTransaction transaction =
getFragmentManager().beginTransaction();

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

```
// цього фрагмента, і додати транзакцію в кінець стека

transaction.replace(R.id.fragment_container, newFragment);

transaction.addToBackStack(null);

// Зафіксувати транзакцію

transaction.commit();
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    }
    ...
}
```

Тоді операція, яка містить цей фрагмент, реалізує інтерфейс `OnArticleSelectedListener` та перевизначає метод `onArticleSelected()`, щоб сповіщати фрагмент 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 при додаванні фрагмента в операцію) створює екземпляр класу `OnArticleSelectedListener`, виконавши приведення типу об'єкта `Activity`, який передається методу `onAttach()`:

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

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

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

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

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

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

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

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

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

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

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

1) відновлений (Resumed) – фрагмент видно під час виконання операції;

2) призупинений (Paused) – на передньому плані виконується і знаходиться в фокусі інша операція, але операція, яка містить даний фрагмент, як і раніше, видна (операція переднього плану частково прозора або не займає весь екран);

3) зупинений (Stopped).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

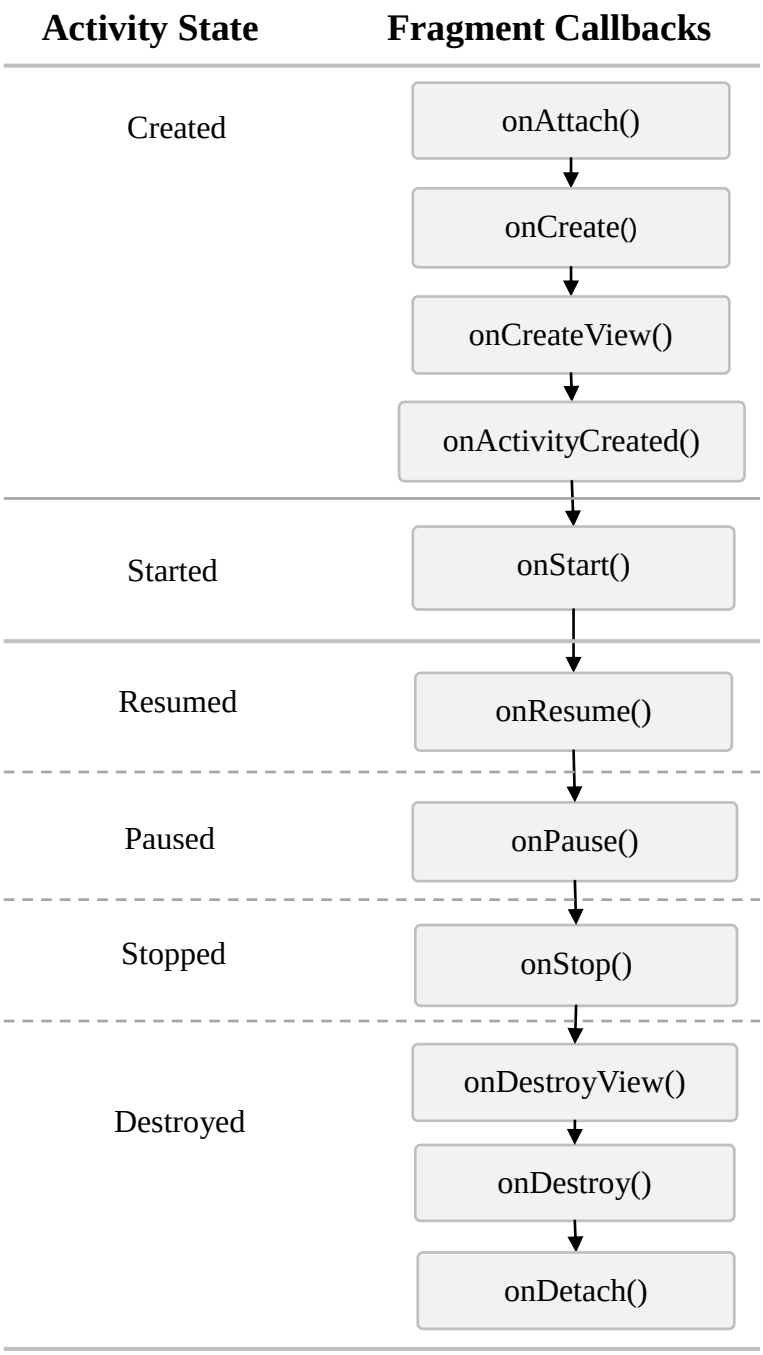

Рисунок 2.12 – Вплив життєвого циклу операції на життєвий цикл фрагмента

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

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

Примітка. Повний вихідний код цієї операції знаходиться в файлі FragmentLayout.java.

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

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

Тут застосовується макет `fragment_layout.xml`:

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

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

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

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

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

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

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

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

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

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

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

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

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

```
            //вибраний елемент.

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

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

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

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

                // Виконуємо транзакцію, замінивши будь-який існуючий фрагмент
                // з цим всередині рамки.
                FragmentTransaction ft =
getFragmentManager().beginTransaction();
                if (index == 0) {
                    ft.replace(R.id.details, details);
                } else {
                    ft.replace(R.id.a_item, details);
                }
                ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
                ft.commit();
            }

        } else {
            // В іншому випадку нам потрібно запустити нову дію для
      // відображення фрагмента діалогу з виділеним текстом.
            Intent intent = new Intent();
            intent.setClass(getActivity(), DetailsActivity.class);
```

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

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

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

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

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

        return f;
    }

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

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

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

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

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

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

```
public static class DetailsActivity extends Activity {

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

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

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

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

```
        }
    }
}
```

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

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

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

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

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

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

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

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

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

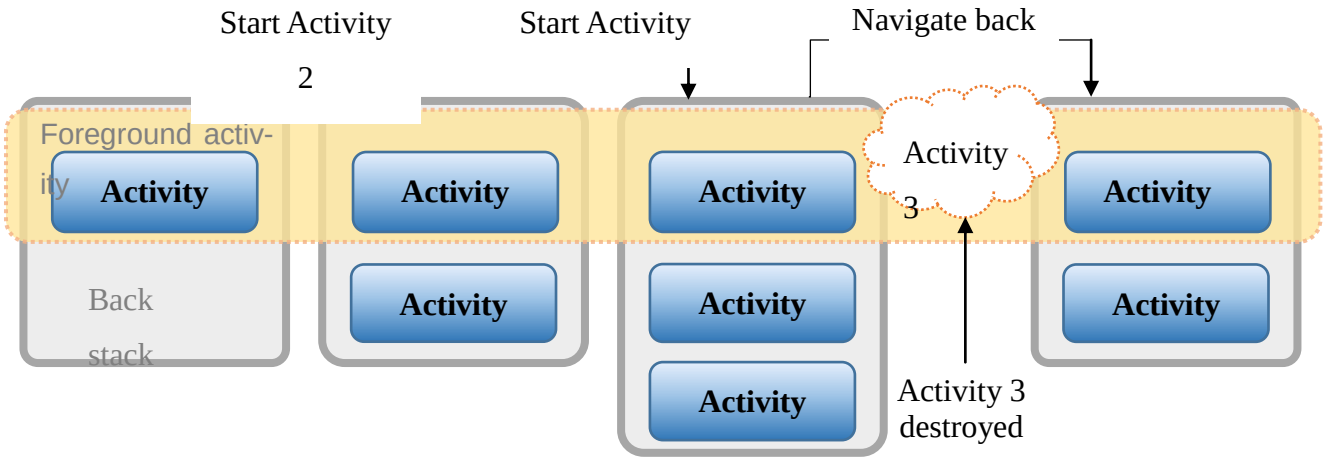

Рисунок 2.13 – Стан операцій і поточний стан стека переходів назад в кожен момент часу

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

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

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

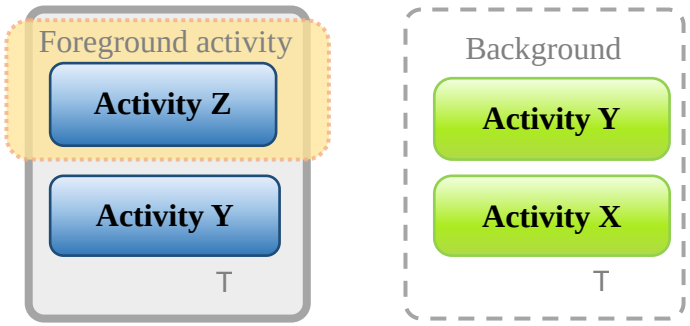

Рисунок 2.14 – Фоновий та активний режими задач

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

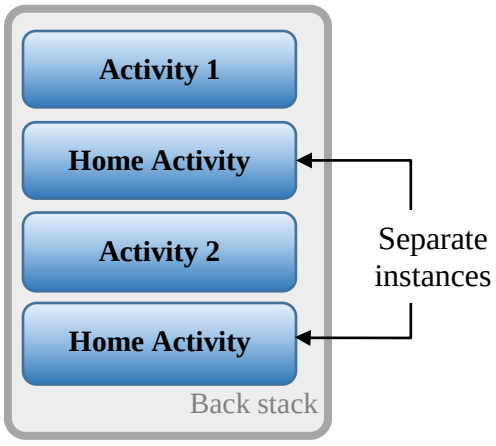

Рисунок 2.15 – Створення кількох примірників однієї операції

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

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

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

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

Підіб'ємо підсумки поведінки операцій і задач:

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

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

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

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

2.3.1. Збереження стану операції

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

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

2.3.2. Управління задачами

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

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

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

– `taskAffinity;`

– `launchMode;`

– `allowTaskReparenting;`
– `clearTaskOnLaunch;`
– `alwaysRetainTaskState;`
– `finishOnTaskLaunch.`

Головні прапорці намірів, які можна використовувати – це:

– `FLAG_ACTIVITY_NEW_TASK;`
– `FLAG_ACTIVITY_CLEAR_TOP;`
– `FLAG_ACTIVITY_SINGLE_TOP.`

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

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

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

2.3.3. Визначення режимів запуску

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

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

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

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

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

2.3.4. Використання файлу маніфесту

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

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

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

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

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

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

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

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

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

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

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

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

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

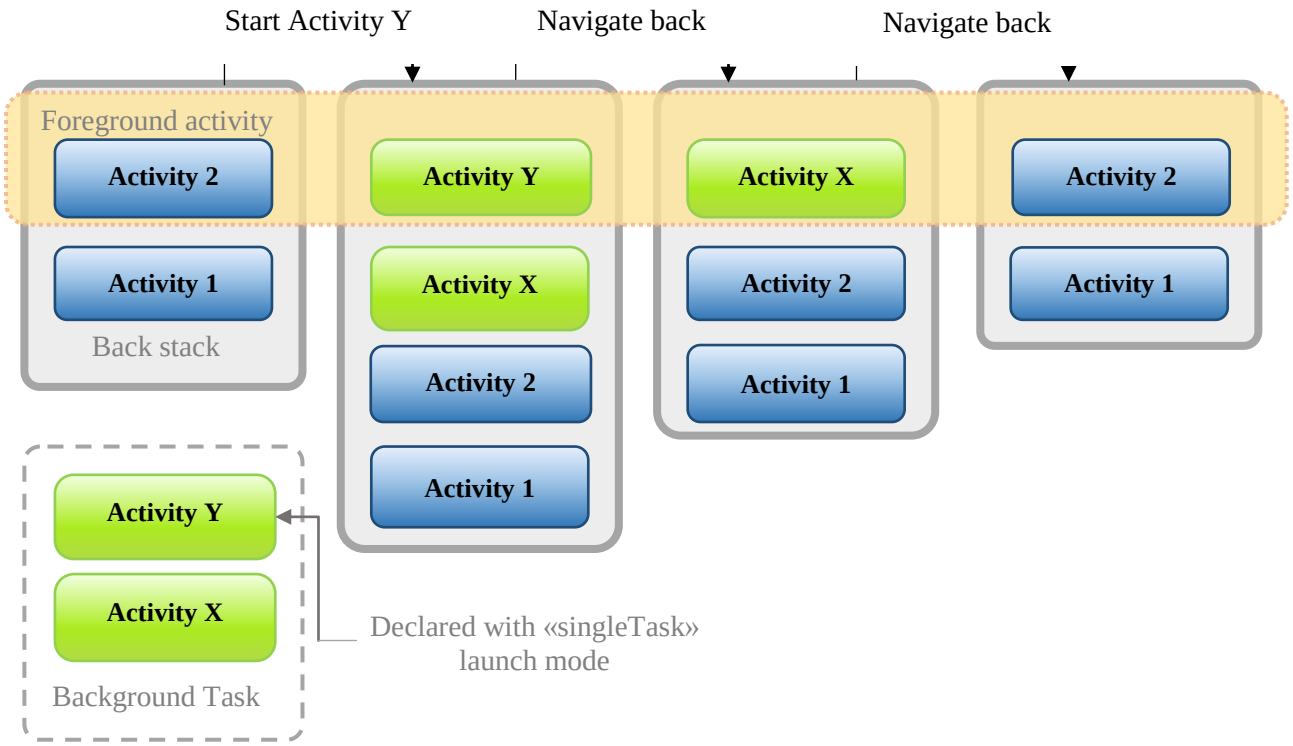

Рисунок 2.16 – Сценарій «singleInstance»

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

2.3.5 Використання прапорців намірів

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

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

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

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

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

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

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

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

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

Прив'язка вступає в гру в двох випадках:

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

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

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

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

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

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

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

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

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

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

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

2.3.6. Запуск задачі

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

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

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

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

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

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

2.3.7. Екран огляду (Overview screen)

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

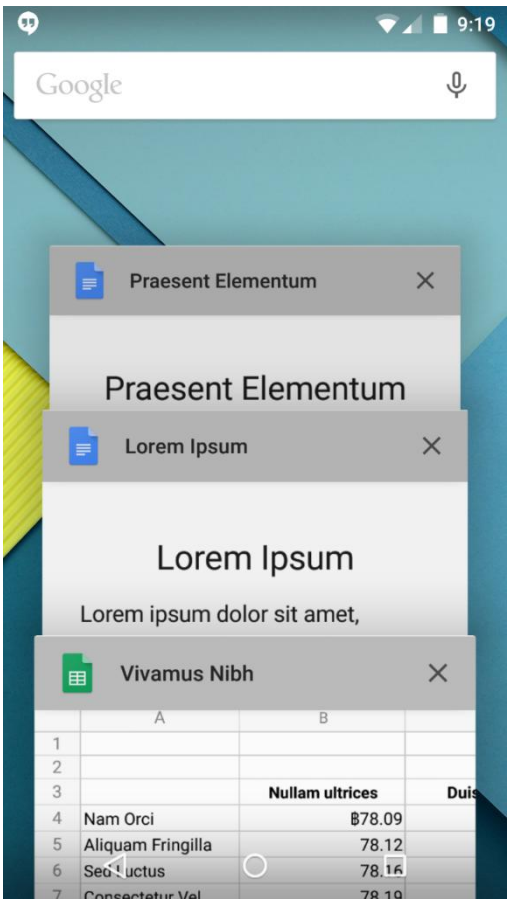

Рисунок 2.17 – Екран огляду, на якому показані три документи Google Диска, подані у вигляді окремих задач

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

2.3.8. Додавання задач на екран огляду

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

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

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

DocumentCentricActivity.java

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

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

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

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

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

```
NewDocumentActivity.java

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

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

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

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

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

FLAG_ACTIVITY_MULTIPLE_TASK.

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

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

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

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

2.3.9. Видалення задач

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4.1. Типи об'єктів Intent

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

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

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

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

2. Операція A створює об’єкт `Intent` з описом дії і передає його до методу `startActivity()`.

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

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

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

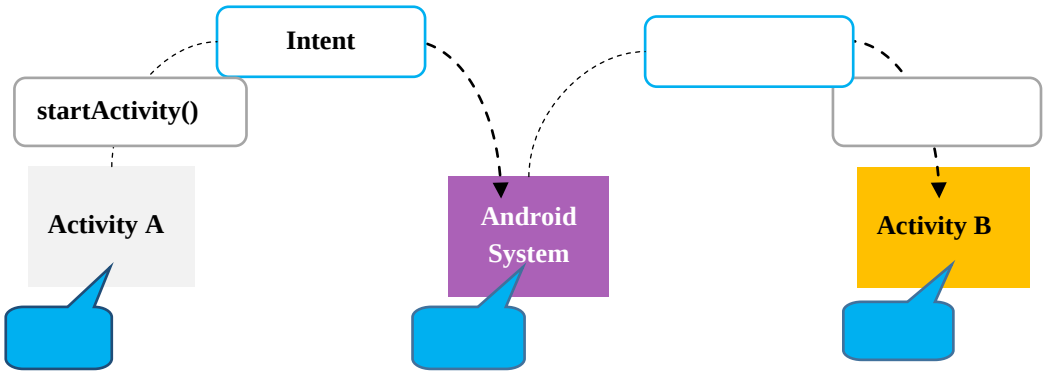

Рисунок 2.18 – Процес передачі неявного об’єкта `Intent` системою для запуску іншої операції

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

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

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

2.4.2. Створення об'єкта Intent

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`CATEGORY_BROWSABLE`

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

`CATEGORY_LAUNCHER`

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

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

Вказати категорію можна за допомогою `addCategory()`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

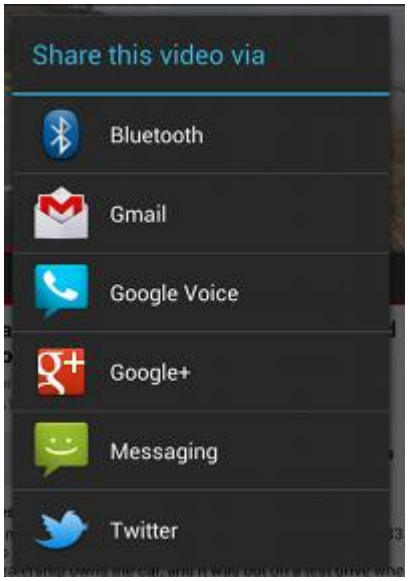

Рисунок 2.19 – Діалогове вікно вибору

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

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

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

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


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

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

2.4.3. Отримання неявного об'єкта Intent

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4.4. Використання очікувального об'єкта Intent

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

Основні варіанти використання очікувального об'єкта `Intent`:

– оголошення об'єкта `Intent`, який повинен буде виконуватися, коли користувач виконуватиме дію з вашим повідомленням (`NotificationManager` системи Android виконує `Intent`);

– оголошення об'єкта `Intent`, який повинен буде виконуватися, коли користувач виконуватиме дію з вашим віджетом додатка (додаток головного екрану виконує `Intent`);

– оголошення об'єкта `Intent`, який повинен буде виконуватися в зазначений час в майбутньому (`AlarmManager` системи Android исполняет `Intent`).

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

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

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

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

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

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

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

2.4.5. Дозвіл об'єктів Intent

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

– дія об'єкта `Intent`;

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

– категорія об'єкта `Intent`.

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

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

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

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

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

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

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

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

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

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

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

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

```
scheme ://host :port /path.
```

Наприклад:

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

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

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

– якщо схему не вказано, вузол ігнорується;

– якщо вузол не вказано, порт ігнорується;

– якщо не вказано ні схему, ні вузол, шлях ігнорується.

Коли URI, вказанний в объекті Intent, порівнюється з URI з фільтра, порівнювання виконується тільки з тими складовими частинами URI, які наведені у фільтрі. Наприклад:

– якщо у фільтрі вказаноа тільки схему, то всі URI до цієї схеми будуть відповідати фільтру;

– якщо у фільтрі вказано схему і повноваження, але відсутній шлях, всі URI з такими ж схемою і повноваженнями пройдуть фільтр, а їх шляхи враховуватися не будуть;

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

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

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

1. Об'єкт Intent, який не містить ні URI, ні тип MIME, пройде цей тест, тільки якщо у фільтрі не вказано жодних URI або типів MIME.

2. Об'єкт Intent, в якому є URI, але відсутній тип MIME (ні явний, ні той, який можна вивести з URI), пройде цей тест, тільки якщо URI відповідає формату URI з фільтра, а у фільтрі також не вказано тип MIME.

3. Об'єкт Intent, в якому є тип MIME, але відсутній URI, пройде цей тест, тільки якщо у фільтрі вказаний той же тип MIME і не вказано формат URI.

4. Об'єкт Intent, в якому є і URI, і тип MIME (явний чи той, який можна вивести з URI), пройде тільки частину цього тесту, яка перевіряє тип MIME, в тому випадку, якщо цей тип збігається з типом, наведеним у фільтрі. Він прой-

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

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

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

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

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

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

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

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

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

2.5 Спливаючі повідомлення

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

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

Рисунок 2.20 – Спливаюче вікно «Чернетку збережено»

2.5.1. Основи

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Запитання для самоконтролю

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

3 СЛУЖБИ І СЕРВІСИ МОБІЛЬНИХ ПЛАТФОРМ

Програмний компонент: Служби (Services). `Service` є компонентом програми, який може виконувати тривалі операції у фоновому режимі і який не містить користувацького інтерфейсу. 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 і навіть виконувати міжпроцесну взаємодію (IPC). Наприклад, служба може обробляти мережні транзакції, відтворювати музику, виконувати введення-виведення файлу або взаємодіяти з постачальником контенту, і все це у фоновому режимі.

Фактично служба може прибирати дві форми:

1. Запущену. 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 був знищений компонент, який її запустив. Зазвичай запущена служба виконує одну операцію і не повертає результатів викликаючому компоненту. Коли операція виконана, служба має зупинитися самостійно.

2. Прив'язану. 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 і навіть робити це між різними процесами за допомогою міжпроцесної взаємодії (IPC). Прив'язана служба працює до тих пір, поки до неї прив'язаний інший компонент програми. До служби можуть бути прив'язані кілька функцій одночасно, але коли вони скасовують прив'язку, 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 може допускати прив'язку. Це залежить від реалізації пари методів зворотного виклику: `onStartCommand()` компонентів дозволяє запускати служби, а `onBind()` – виконувати прив'язку.

Незалежно від стану програми (запущена, прив'язана або обидва 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 і заблокувати доступ до неї з інших додатків.

Увага! Служба працює в основному потоці головного процесу – служба не створює свого потоку і не виконується в окремому процесі. Це означає, що якщо ваша служба збирається виконувати будь-яку роботу з високим навантаженням ЦП або блокуючі операції (наприклад, відтворення MP3 або мережеві операції), необхідно створити в службі новий потік для виконання цієї роботи. Використовуючи окремий потік, знижується ризик виникнення помилок «Додаток не відповідає», і основний потік програми може відпрацьовувати взаємодію користувача з вашими операціями.

3.1. Основи

Щоб створити службу, необхідно створити підклас класу `Service` (або одного з існуючих його підкласів). У вашій реалізації необхідно перевизначити деякі методи зворотного виклику, які обробляють ключові моменти життєвого циклу служби та за необхідності надають механізм прив'язування компонентів. Найбільш важливими методами зворотного виклику, які необхідно перевизначити, є:

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

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

– `onCreate()`. Система викликає цей метод при першому створенні служби для виконання одноразових процедур налаштування (перед викликом `onStartCommand()` або `onBind()`). Якщо служба вже запущена, цей метод не викликається.

– `onDestroy()`. Система викликає цей метод, коли служба більше не використовується та коли виконується її знищення. Ваша служба повинна реалізувати це для очищення ресурсів, таких, як потоки, зареєстровані приймачі, ресивери і т. д. Це останній виклик, який отримує служба.

Якщо компонент запускає службу за допомогою виклику `startService()` (що приводить до виклику `onStartCommand()`), то служба продовжує роботу, поки вона не зупиниться самостійно з допомогою `stopSelf()` або поки 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'язаний компонент. Як тільки виконується скасування прив'язки служби до всіх клієнтів, система знищує службу.

Система Android буде примусово зупиняти службу тільки в тому випадку, коли не вистачає пам'яті і коли необхідно відновити системні операції, які відображаються на передньому плані. Якщо служба прив'язана до операції, яка відображається на передньому плані, менш ймовірно, що вона буде знищена, і якщо служба оголошена для виконання у фоновому режимі (як обговорювалося вище), вона майже ніколи не буде знищуватися. В іншому випадку, якщо служба була запущена і є тривалою, система з часом буде опускати її положення в списку фонових завдань, і служба стане дуже чутливою до знищення – якщо ваша служба працює, то ви повинні передбачити витончену обробку її перезапуску системою. Якщо система знищує вашу службу, вона запускає її, як тільки знову з'являється доступ до ресурсів (хоча це також залежить від значення, що повертається методом `onStartCommand()`, як обговорюється нижче).

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

3.2. Що краще – служба чи потік?

Служба – це просто компонент, який може виконуватися у фоновому режимі, навіть коли користувач не взаємодіє з додатком. Отже, необхідно 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, ймовірно, слід створити новий потік, а не службу. Наприклад, якщо ви бажаєте відтворювати певну музику, але тільки під час роботи, операції, то необхідно створити потік в `onCreate()`, запустити його виконання в методі `onStart()`, а потім зупинити його в методі `onStop()`. Також розгляньте можливість використання класу AsyncTask або HandlerThread замість звичайного класу `Thread`. У документі «Процеси і потоки» міститься додаткова інформація про ці потоки.

Пам’ятайте, що якщо ви використовуєте службу, вона виконується в основному потоці вашого додатка за замовчуванням, тому потрібно створити новий потік в службі, якщо вона виконує інтенсивні або блокувальні операції.

3.2.1. Оголошення служби в маніфесті

Всі служби, як і операції (та інші компоненти), повинні бути оголошені в файлі маніфесту вашого додатка.

Щоб оголосити службу, додайте елемент `<service>`, як дочірній элемент `<application>`. Наприклад:

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

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

Є інші атрибути, які можна включити в елемент `<service>` для завдання властивостей, наприклад, необхідних для запуску дозволів, і процесу, в якому повинна виконуватися служба. Атрибут `android:name` є єдиним обов’язковим атрибутом – він вказує ім’я класу для служби. Після публікації вашого додатка вам не слід змінювати це ім’я, оскільки це може зруйнувати код через залежність від явних намірів, використовуваних, щоб запустити або прив’язати службу (ознайомтеся з публікацією «Речі, які не можна змінювати» в блозі розробників).

Для забезпечення безпеки додатка **завжди використовуйте явний намір при запуску або прив’язці** `Service` і не розкривайте фільтри намірів для служби. Якщо вам важливо допустити деяку невизначеність щодо того, яка служба запускається, то можна надати фільтри для ваших намірів служб і виключити ім’я компонента з `Intent`, але потім необхідно встановити пакет для наміру за допомогою `setPackage()`, який забезпечує достатнє усунення неоднозначності для цільової служби.

Додатково можна забезпечити доступність вашої служби тільки для вашого додатка, включивши атрибут `android:exported` і встановивши для нього значення «`false`». Це не дозволяєить іншим програмам запускати вашу службу навіть при використанні явного наміру.

3.3. Створення запущеної служби

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

При запуску служба має термін життя, що не залежить від компонента, що її запустив, і може працювати у фоновому режимі протягом необмеженого часу, навіть якщо був знищений компонент, який її запустив. Тому після виконання своєї роботи служба повинна 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 вказує службу і будь-які дані, які служба повинна використовувати. Служба отримує цей об'єкт `Intent` в методі `onStartCommand()`.

Припустимо, що операції потрібно зберегти певні дані в мережній базі даних. Операція може запустити службу та надати їй дані для збереження, передавши намір метод `startService()`. Служба отримує намір в методі `onStartCommand()`, підключається до Інтернету і виконує транзакцію з базою даних. Коли транзакція виконана, служба зупиняється самостійно і знищується.

Увага! За замовчуванням служби працюють в тому ж процесі, що і додаток, в якому вони оголошені, а також в основному потоці цього додатка. Тому, якщо ваша служба виконує інтенсивні або блокувальні операції, в той час як користувач взаємодіє з операцією з того ж додатка, служба буде сповільнювати виконання операції. Щоб уникнути негативного впливу на швидкість роботи програми, необхідно запустити новий потік всередині служби.

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

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

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

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

3.3.1. Спадкування класу IntentService

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

Клас `IntentService` виконує такее:

1. Створює робочий потік за замовчуванням, який виконує всі наміри, доставлені в метод `onStartCommand()`, окремо від основного потоку вашого додатка.

2. Створює робочу чергу, яка передає наміри по одному в вашу реалізацію методу `onHandleIntent()`, тому не треба турбуватися щодо багатопоточності.

3. Зупиняє службу після обробки всіх запитів запуску, тому вам ніколи не потрібно викликати `stopSelf()`.

4. Здійснює реалізацію методу `onBind()`, яка повертає `null`.

5. Здійснює реалізацію методу `onStartCommand()` за замовчуванням, яка відправляє намір в робочу чергу і потім у вашу реалізацію `onHandleIntent()`.

Все це означає, що вам достатньо реалізувати метод `onHandleIntent()` для виконання роботи, наданої клієнтом (хоча, крім того, ви повинні надати маленький конструктор для служби).

Все, що потрібно: конструктор і реалізація класу `onHandleIntent()`.

Якщо необхідно перевизначити також і інші методи зворотного виклику, такі, як `onCreate()`, `onStartCommand()` або `onDestroy()`, обов'язково викличте реалізацію суперкласу, щоб клас `IntentService` міг правильно обробляти життєвий цикл робочого потоку.

Далі наведено приклад реалізації класу `IntentService`:

```
public class HelloIntentService extends IntentService {
  /**
* Потрібен конструктор, і повинен викликати IntentService(String)
* конструктор з ім'ям для робочого потоку.
   */
  public HelloIntentService() {
      super("HelloIntentService");
  }

  /**
     * IntentService викликає цей метод з робочого потоку за за-
мовчуванням.
* Коли цей метод виконався, IntentService зупиняє службу,
* при необхідності.
   */
```

```
  @Override
  protected void onHandleIntent(Intent intent) {
//Як правило, ми будемо виконувати деяку роботу тут, наприклад,
скачати 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) {
              }
          }
      }
  }
}
```

Наприклад, метод `onStartCommand()` повинен повертати реалізацію за замовчуванням (яка доставляє намір `onHandleIntent()`).

```
@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);
}
```

Крім `onHandleIntent()`, єдиним методом, з якого вам не потрібно викликати суперклас, є метод `onBind()` (але його потрібно реалізовувати тільки у випадку, якщо ваша служба допускає прив'язку).

3.3.2. Спадкування класу Service

Як було зазначено в попередньому розділі, використання класу IntentService значно спрощує реалізацію запущеної служби. Проте, якщо необхідно, щоб ваша служба підтримувала багатопоточність (замість обробки запитів запуску через робочу чергу), можна успадковувати клас `Service` для обробки кожного наміру.

Як приклад наведено наступну реалізацію класу `Service`, яка виконує ту ж роботу, що і клас `IntentService`, який для кожного запиту запуску використовує робочий потік для виконання задачі і обробляє запити по одному:

```
  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,
скачати 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) {
                  }
              }
          }
// Зупиняємо сервіс за допомогою startId, таким чином ми не
//зупиняємо сервіс у середині обробки іншої роботи
  stopSelf(msg.arg1);
      }
  }
  @Override
  public void onCreate() {
// Запуск потоку, в якому виконується служба. Зверніть увагу, що
ми
// створюємо окремий поток, тому що служба зазвичай працює у
// основному потоці процесу, який ми не хочемо заблокувати. Ми
// також задали йому фоновий пріоритет, так що напружена робота
не                    //порушить нашого користувальницького інте-
рфейсу.
    HandlerThread thread = new
HandlerThread("ServiceStartArguments",
            Process.THREAD_PRIORITY_BACKGROUND);
    thread.start();
// Отримуємо HandlerThread і використовувати його для нашого об-
робника
    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
почати
```

```
// роботу і відправляємо ID, так що ми знаємо, який запит ми зу-
пинили, коли ми 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’язку, тому повернутаємо null
      return null;
  }

  @Override
  public void onDestroy() {
    Toast.makeText(this, "service done",
Toast.LENGTH_SHORT).show();
  }
}
```

Як можна бачити, цей код значно довший, ніж код з використанням класу `IntentService`.

Однак через те що кожен виклик `onStartCommand()` обробляється самостійно, можна виконувати декілька запитів одночасно. Даний код виконує не зовсім цю роботу, але за необхідності можна створювати нові потоки для кожного запиту і відразу запускати їх (а не чекати завершення попереднього запиту).

Зверніть увагу, що метод `onStartCommand()` повинен повертати ціле число. Це ціле число описує, як система має продовжувати виконання служби у разі, коли система знищила її (як описано вище, реалізація за замовчуванням для класу `IntentService` обробляє цю ситуацію, хоча був змінений хід реалізації). Значення, що повертається методом `onStartCommand()`, має бути однією з наступних констант:

– `START_NOT_STICKY`. Якщо система знищує службу після повернення з `onStartCommand()`, *не потрібно* повторно створювати службу, якщо немає намірів, що очікують доставку. Це найбезпечніший варіант, він дозволяє уникнути запуску вашої служби, коли це не потрібно і коли ваш додаток може просто перезапустити будь-які незавершені задачі.

– `START_STICKY`. Якщо система знищує службу після повернення з `onStartCommand()`, створіть службу і викличте `onStartCommand()`, але не передавайте останнім намір повторно. Замість цього система викликає метод `onStartCommand()` з наміром, якеий має значення `null`, якщо немає очікува-

льних намірів для запуску служби. Якщо очікувальні наміри є, вони доставляються. Це підходить для мультимедійних програвачів (або подібних служб), які не виконують команди, а працюють незалежно і чекають завдання.

– `START_REDELIVER_INTENT`. Якщо система знищує службу після повернення з `onStartCommand()`, повторно створіть службу і викличте `onStartCommand()` з останнім наміром, який був доставлений в службу. Всі очікуючі наміри доставляються по черзі. Це підходить для служб, що активно виконують задачу, яка має бути відновленао негайно, наприклад, для завантаження файлу.

Для отримання додаткових відомостей див. довідкову документацію за посиланням для кожної константи.

3.3.3. Запуск служби

Можна запустити службу операції або іншого компонента, передавши об'єкт `Intent` (вказує службу, яку потрібно запустити) в `startService()`. Система Android викликає метод `onStartCommand()` служби і передає їй `Intent`. (Ні в якому разі не слід викликати метод `onStartCommand()` напряму).

Наприклад, операція може запустити службу з прикладу у попередньому розділі (`HelloSevice`), використовуючи явний намір за допомогою `startService()`:

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

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 подає рив'язку, то намір, що доставляється за допомогою `startService()`, є єдиним режимом зв'язку між компонентом програми та службою. Однак якщо необхідно, щоб служба відправляла результат назад, клієнт, який запускає службу, може створити об'єкт `PendingIntent` для повідомлення (з допомогою `getBroadcast()`) і доставити його на службу в об'єкті `Intent`, який запускає службу. Потім служба може використовувати повідомлення для доставки результату.

Кілька запитів запуску служби приводять до кількох відповідних викликів методу `onStartCommand()` служби. Однак для її зупинки достатньо тільки одного запиту на зупинку служби (з допомогою `stopSelf()` або `stopService()`).

3.3.4. Зупинка служби

Запущена служба повинна керувати своїм життєвим циклом. Тобто система не зупиняє і не знищує службу, якщо не потрібно відновити пам'ять системи, і служба продовжує роботу після повернення з методу `onStartCommand()`. Тому служба повинна зупинятися самостійно за допомогою виклику методу `stopSelf()`, або інший компонент може зупинити її за допомогою виклику методу `stopService()`.

Отримавши запит на зупинку за допомогою `stopSelf()` або `stopService()`, система якомога швидше знищує службу.

Однак якщо служба обробляє кілька запитів `onStartCommand()` одночасно, то не треба зупиняти службу після завершення обробки запиту запуску, оскільки, ймовірно, вже був отриманий 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 ваш запит на зупинку служби завжди заснований на самому останньому запиті запуску. Тобто коли ви викликаєте `stopSelf(int)`, передається ідентифікатор запиту запуску (ідентифікатор `startId`, доставлений в `onStartCommand()`), якому відповідає ваш запит зупинки.

Увага! Ваш додаток обов'язково повинен зупиняти свої служби по закінченні роботи, щоб уникнути витрачання ресурсів і споживання енергії акумулятора. За необхідності інші компоненти можуть зупинити службу за допомогою виклику методу `stopService()`. Навіть якщо можна виконувати прив'язку служби, слід завжди зупиняти службу самостійно, якщо вона колинебудь отримувала виклик `onStartCommand()`.

3.4. Створення прив'язаної служби

Прив'язана служба – це служба, яка допускає прив'язку до неї компонентів програми за допомогою виклику `bindService()` для створення довготривалого з'єднання (і зазвичай не дозволяє компонентам запускати її за допомогою виклику `startService()`).

Прив'язана служба створюється, коли треба взаємодіяти зі службою операцій та з іншими компонентами вашого додатка чи показувати деякі функції вашого додатка іншим додаткам 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 ви-

кликати метод `bindService()` для вилучення інтерфейсу і почати викликати методи служби. Служба існує тільки для обслуговування прив'язаного до неї компонента, а тому, коли немає компонентів, прив'язаних до служби, система знищує її (вам не потрібно зупиняти прив'язану службу, як це вимагається для служби, запущеної за допомогою `onStartCommand()`).

Щоб створити прив'язану службу, необхідно в першу чергу визначити інтерфейс взаємодії клієнта зі службою. Цей інтерфейс між службою та клієнтом повинен бути реалізацією об'єкта `IBinder`, яку ваша служба повинна повертати з методу зворотного виклику `onBind()`. Після того як клієнт отримає об'єкт `IBinder`, він може почати взаємодію зі службою за допомогою цього інтерфейсу.

Одночасно до служби можуть бути прив'язані кілька клієнтів. Коли клієнт закінчує взаємодію зі службою, він викликає `unbindService()` для скасування прив'язки. Як тільки не залишається ні одного клієнта, прив'язаного до служби, система знищує службу.

Існує декілька способів реалізації прив'язаної служби, і ці реалізації складніші, ніж реалізації запущеної служби, тому обговорення прив'язаної служби наведено в окремому документі «Прив'язані служби».

3.5. Відправка повідомлень користувачеві

Після запуску служба може повідомляти користувача про події, використовуючи спливаючі попередження або повідомлення в рядку стану.

Спливаюче повідомлення – це повідомлення, що короткочасно з'являється на поточному вікні, тоді як повідомлення в рядку стану – це значок в 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 файлу), і користувач може діяти. Коли користувач вибирає повідомлення в розширеному вигляді, повідомлення може запустити операцію (наприклад, для перегляду завантаженого файлу).

Додаткову інформацію див. у посібнику для розробників «Спливаючі повідомлення» і «Повідомлення в рядку стану».

3.6. Запуск служби на передньому плані

Служба переднього плану – це служба, про яку користувач активно обізнаний, і тому вона не є кандидатом для видалення системою в разі нестачі

пам’яті. Служба переднього плану повинна виводити повідомлення в рядок стану, який знаходиться під заголовком «Постійні». Це означає, що повідомлення не може бути видалене, поки служба не буде зупинена або видалена з переднього плану.

Наприклад, музичний програвач, який відтворює музику зі служби, має бути налаштований на роботу на передньому плані, тому що користувач точно знає про його роботу. Повідомлення в рядку стану може показувати поточну музичну композицію і дозволяти користувачу запускати операцію для взаємодії з музичним програвачем.

Для запиту на виконання вашої служби на передньому плані викличте метод `startForeground()`. Цей метод має два значення: ціле число, яке однозначно ідентифікує повідомлення, і об’єкт `Notification` для рядка стану. Наприклад:

```
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);
```

Увага! Цілочисельний ідентифікатор `ID`, який передається у метод `startForeground()`, не повинен дорівнювати 0.

Щоб видалити службу з переднього плану, викличте метод `stopForeground()`, що містить логічне значення, яке вказує, чи потрібно видаляти повідомлення в рядку стану. Цей метод не зупиняє службу. Однак якщо зупиняється служба, що працює на передньому плані, повідомлення буде видалено.

3.7. Управління життєвим циклом служби

Життєвий цикл служби набагато простіший, ніж життєвий цикл операції. Однак набагато важливіше приділити пильну увагу тому, як ваша служба створюється і знищується, тому що служба може працювати у фоновому режимі без відома користувача.

Життєвий цикл служби, від створення до знищення, може слідувати двома різними шляхами, а саме:

1. Запущена служба. Служба створюється, коли інший компонент викликає метод `startService()`. Потім служба працює протягом необмеженого часу і має зупинитися самостійно за допомогою виклику методу `stopSelf()`. Інший компонент також може зупинити службу допомогою виклику методу `stopService()`. Коли служба зупиняється, система знищує її.

2. 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 можуть бути прив'язані кілька клієнтів, і коли вони скасовують прив'язку, система знищує службу (служба не повинна зупинятися самостійно).

Ці дві служби необов'язково працюють незалежно одна від одної. Тобто можна прив'язати службу, яка вже була запущена за допомогою методу `startService()`. Наприклад, фонова музична служба може бути запущена 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'язку до служби за допомогою виклику методу `bindService()`. У подібних випадках методи `stopService()` і `stopSelf()` фактично не зупиняють службу, поки не буде скасована прив'язка всіх клієнтів.

3.8. Реалізація зворотних викликів життєвого циклу

Подібно до операції, служба містить методи зворотного виклику життєвого циклу, які можна реалізувати для контролю змін стану служби і виконання роботи у відповідні моменти часу. Зазначена нижче базова служба показує кожний з методів життєвого циклу:

```
public class ExampleService extends Service {
    int mStartMode;       // визнаємо що робити, якщо сервіс не
працює
    IBinder mBinder;      // інтерфейс для клієнтів
    boolean mAllowRebind; // визначаємо, чи слід використовувати
onRebind
    @Override
    public void onCreate() {
        // Сервіс створений
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int
startId) {
        // сервіс запускається
        return mStartMode;
    }
```

```
    @Override
    public IBinder onBind(Intent intent) {
        //Клієнт прив’язується до сервіса 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 сервіса за допомогою
bindService()
        // після onUnbind(), що вже був викликаний
}
    @Override
    public void onDestroy() {
        // Послуга більше не використовується і знищується
  }
}
```

Примітка. На відміну від методів зворотного виклику життєвого циклу операції, вам не потрібно викликати реалізацію суперкласу цих методів зворотного виклику.

На рис. 3.1 зліва показаний життєвий цикл, коли служба створена за допомогою методу `StartService()`, а на справа – життєвий цикл, коли служба створена за допомогою методу `bindService()`.

За допомогою реалізації цих методів можна відстежувати два вкладених цикли в життєвому циклі служби:

- **Весь життєвий цикл** служби, який відбувається між викликом методу `onCreate()` і поверненням з методу `onDestroy()`. Подібно до операції, служба виконує початкову настройку 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()`, а потім зупинити потік в методі `onDestroy()`.

Методи `onCreate()` і `onDestroy()` викликаються для всіх служб, незалежно від методу створення: `startService()` чи `bindService()`.

- **Активний життєвий цикл** служби, який починається з виклику методу `onStartCommand()` чи `onBind()`. Кожен метод спрямовується наміром `Intent`, який був переданий методу `startService()` чи `bindService()` відповідно.

Якщо служба запущена, активний життєвий цикл закінчується одночасно із закінченням всього життєвого циклу (служба активна навіть після поверне-

ня з методу `onStartCommand()`). Якщо служба є прив'язаною, активний життєвий цикл закінчується, коли повертається метод `onUnbind()`.

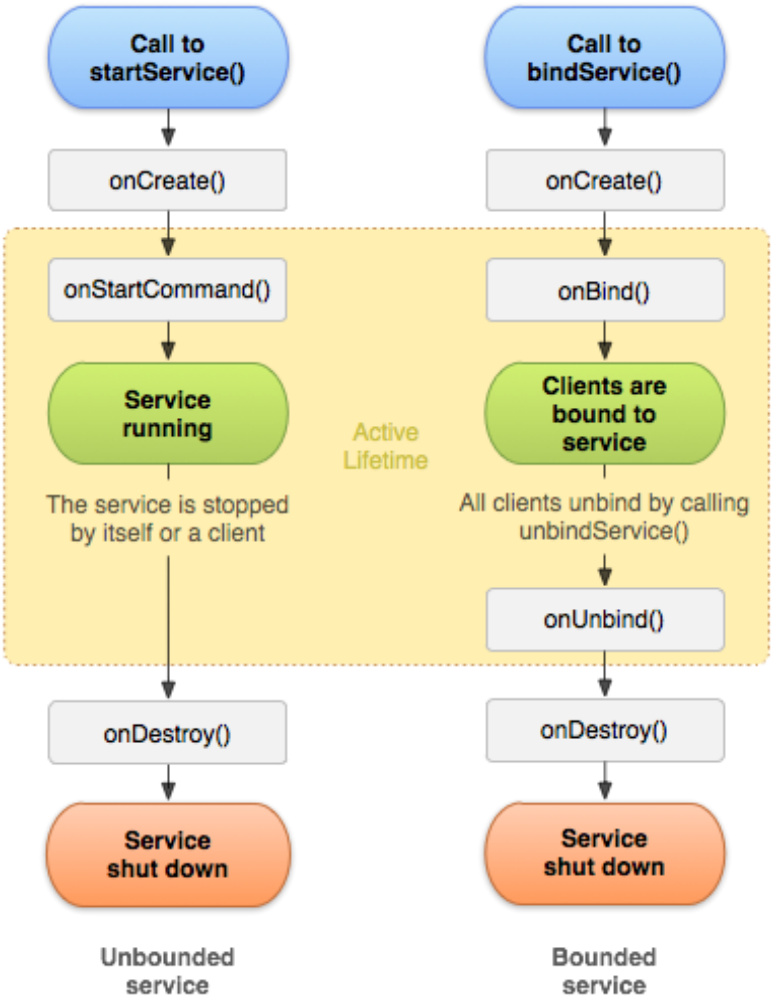

Рисунок 3.1 – Життєвий цикл служби

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 при зупинці служби – метод `onDestroy()` є єдиним одержуваним методом зворотного виклику.

На рис. 3.1. справа проілюстровано типові методи зворотного виклику для служби. Хоча на цьому рисунку служби, створені за допомогою методу `startService()`, відокремлені від служб, створених за допомогою методу `bindService()`, пам'ятайте, що будь-яка служба, незалежно від способу запуску, дозволяє клієнтам виконувати прив'язку до неї. Тому служба, спочатку створена за допомогою методу `onStartCommand()` (клієнтом, який викликав `startService()`), може отримувати виклик методу `onBind()` (коли клієнт викликає метод `bindService()`).

3.9. 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 і 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.

У цьому документі розповідається, як створити прив’язану службу, включаючи прив’язку служби до інших компонентів програми.

3.9.1. Основи

Прив’язана служба являє собою реалізацію класу `Service`, яка дозволяє іншим програмам прив’язуватися до нього і взаємодіяти з ним. Щоб забезпечити прив’язку 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.

3.9.2. Прив’язка до запущеної служби

Можна створити службу, яка одночасно і запущена, і прив’язана. 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’язку служби, то після її запуску система *не* знищує її після скасування всіх прив’язок 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()`, в деяких випадках потрібно реалізувати обидва ці методи. Наприклад, в музичному програвачі може виявитися корисним дозволити виконання служби протягом необмеженого часу, а також забезпечити її прив’язку. Таким чином, операція може запустити службу для відтворення музики, яка буде відтворюватися навіть після виходу користувача з програми. Після повернення користувача до додатка операція може скасувати прив’язку до служби, щоб повернути управління відтворенням.

Для прив’язки до служби клієнт може викликати метод `bindService()`. Після прив’язки він повинен надати реалізацію методу `ServiceConnection`,

який служить для відстеження підключення до служби. Метод `bindService()` повертається негайно без значення, проте коли система Android встановлює 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. Система викликає метод `onBind()`, запускається служба для отримання об'єкта `IBinder` тільки при першій прив'язці клієнта. Після цього система видає такий самий об'єкт `IBinder` для будь-яких додаткових клієнтів, які виконують прив'язку, без повторного виклику методу `onBind()`.

Коли скасовується прив'язка останнього клієнта від служби, система знищує службу (якщо тільки служба не була так само запущена методом `startService()`).

Найважливішу роль в реалізації прив'язаної служби відіграє визначення інтерфейсу, який повертає ваш метод зворотного виклику `onBind()`. Існує кілька різних способів визначення інтерфейсу `IBinder` служби. Кожен з них розглядається в наступному розділі.

Створення прив'язаної служби. При створенні служби, що забезпечує прив'язку, потрібен об'єкт `IBinder`, який забезпечує програмний інтерфейс, за допомогою якого клієнти можуть взаємодіяти зі службою. Існує три способи визначення такого інтерфейсу:

1. Розширення класу `Binder`. Якщо служба є приватною і надається в рамках вашого власного додатка, а також виконується в тому ж процесі, що і клієнт (загальний процес), то 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, наявних або в реалізації `Binder`, або навіть в `Service`.

Цей спосіб є кращим, коли служба просто виконується у фоновому режимі для вашого додатка. Цей спосіб не підходить для створення інтерфейсу тільки тоді, коли ваша служба використовується іншими додатками або в окремих процесах.

2. 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`, що відповідає різним типам об'єктів `Message`. Цей об'єкт `Handler` є основою для об'єкта `Messenger`, який, в свою чергу, надає клієнтові об'єкт `IBinder`, завдя-

ки чому останній може відправляти команди в 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 в рамках одного потоку, тому вам не потрібно робити свою службу потокобезпечною.

3. Використання мови AIDL. AIDL (Android Interface Definition Language) виконує всю роботу з розділення об'єктів на примітиви, які операційна система може розпізнати і розподілити по процесах для організації взаємодії між ними (IPC). Попередній спосіб з використанням об'єкта `Messenger` фактично заснований на AIDL, оскільки це його базова структура. Як уже згадувалося вище, об'єкт `Messenger` створює чергу з усіх запитів клієнтів в рамках одного потоку, тому служба одночасно отримує тільки один запит. Однак якщо необхідно, щоб служба обробляла одночасно кілька запитів, можна використовувати AIDL безпосередньо. В такому випадку ваша служба повинна підтримувати багатопоточність і 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, а також забезпечує взаємодію процесів і який в подальшому можна розширити в службі.

Примітка. У більшості додатків **не слід** використовувати AIDL для створення і прив'язки служби, оскільки для цього може знадобитися підтримка багатопоточності, що, в свою чергу, може призвести до більш складної реалізації.

3.9.3. Розширення класу Binder

Якщо ваша служба використовується тільки локальним додатком і не взаємодіє з різними процесами, то можна реалізувати власний клас `Binder`, за допомогою якого клієнт отримує прямий доступ до загальнодоступних методів в службі.

Примітка. Цей варіант підходить тільки в тому випадку, якщо клієнт і служба виконується всередині однієї програми і процесу, що є найбільш поширеною ситуацією. Наприклад, розширення класу відмінно підійде для музичного додатка, в якому необхідно прив'язати операцію до власної служби додатка, яка відтворює музику у фоновому режимі.

Це можна зробити таким чином:

1. Створіть у вашій службі екземпляр класу `Binder`, що має одну з таких характеристик:

- екземпляр містить загальнодоступні методи, які може викликати клієнт;
- екземпляр повертає поточний екземпляр класу `Service`, який містить загальнодоступні методи, які може викликати клієнт;
- екземпляр повертає екземпляр іншого класу, розміщений в службі, що містить загальнодоступні методи, які може викликати клієнт.

2. Поверніть цей екземпляр класу Binder з методу зворотного виклику `onBind()`.

3. У клієнті отримайте клас `Binder` від методу зворотного виклику onServiceConnected () і виконайте виклики до прив'язаної служби за допомогою наданих методів.

Примітка. Служба і клієнт повинні виконуватися в одному і 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 повинні виконуватися в рамках одного і того ж процесу, оскільки цей спосіб не має на увазі будь-якого розподілу за процесами.

Нижче наведено приклад служби, яка надає клієнтам доступ до методів за допомогою реалізації класу `Binder`:

```
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 використовується для клієнта Binder. Тому що ми
знаємо ця послуга завжди
     * працює в тому ж процесі, як її клієнти, ми не повинні 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);
    }
}
```

Об’єкт `LocalBinder` надає клієнтам метод `getService()`, щоб вони могли отримати поточний екземпляр класу `LocalService`. Завдяки цьому клієнти можуть викликати загальнодоступні методи в службі. Наприклад, клієнти можуть викликати метод `getRandomNumber()` зі служби.

Нижче наведено приклад операції, яка виконує прив’язку до класу `LocalService` і викликає метод `getRandomNumber()` при натисканні кнопки:

```
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’язка 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 при натисканні на кнопку (кнопка в файлі макета
      * надає цей метод за допомогою атрибута android:onClick) */
    public void onButtonClick(View v) {
        if (mBound) {
            // Виклик методу з LocalService.
            // Однак, якщо цей виклик був те, що може викликати зависан-
ня, то цей запит повинен
            // відбуваєтися 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, передається 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;
        }
    };
}
```

В наведеному вище прикладі показано, як клієнт прив’язується до служби за допомогою реалізації `ServiceConnection` і зворотного виклику `onServiceConnected()`.

Примітка. В наведеному вище прикладі не виконується явне скасування прив’язки до служби, проте всім клієнтам слід скасовувати прив’язку у відповідних термінах (наприклад, коли операція припиняється).

Приклади коду наведено в статтях, присвячених класам `LocalService.java` й `LocalServiceActivities.java`, в `ApiDemos`.

3.9.4. Використання об’єкта Messenger

Якщо необхідно, щоб служба взаємодіяла з віддаленими процесами, то 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` використовується для створення об’єкта `Messenger` (який є посиланням на об’єкт `Handler`).
- Об’єкт `Messenger` створює об’єкт `IBinder`, який служба повертає клієнтам з методу `onBind()`.
- Клієнти використовують отриманий об’єкт `IBinder` для створення екземпляра об’єкта `Messenger` (який посилається на о бъект `Handler` служби), що використовується клієнтом для відправки об’єктів `Message` в службу.
- Служба отримує кожен об’єкт `Message` в своєму об’єкті `Handler` –

зокрема, в методі `handleMessage()`.

Таким чином, для клієнта відсутні «методи» для відправки виклику служби. Замість цього клієнт відправляє «повідомлення» (об'єкти `Message`), які служба отримує в своєму об'єкті `Handler`.

Нижче наведено приклад служби, яка використовує інтерфейс `Messenger`:

```
public class MessengerService extends Service {
    /** Команда до служби, щоб відобразити повідомлення */
    static final int MSG_SAY_HELLO = 1;

    /**
     * Оброблювач вхідних повідомлень від клієнтів.
     */
    class IncomingHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MSG_SAY_HELLO:
                    Toast.makeText(getApplicationContext(),
"hello!", Toast.LENGTH_SHORT).show();
                    break;
                default:
                    super.handleMessage(msg);
            }
        }
    }

    /**
     * Цільові ми публікуємо для клієнтів для відправки повідомлень
* IncomingHandler.
     */
    final Messenger mMessenger = new Messenger(new
IncomingHandler());

    /**
     * При прив'язці до служби, ми повертаємо інтерфейс до нашого
посланця
     * для відправки повідомлення в службу.
     */
    @Override
    public IBinder onBind(Intent intent) {
        Toast.makeText(getApplicationContext(), "binding",
Toast.LENGTH_SHORT).show();
        return mMessenger.getBinder();
    }
}
```

Зверніть увагу, що метод `handleMessage()` в об'єкті `Handler` – це місце, де служба отримує вхідні об'єкти `Message` і 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()`. Нижче наведено приклад того, як проста операція виконує прив’язку до служби і відправляє їй повідомлення `MSG_SAY_HELLO`:

```
public class ActivityMessenger extends Activity {
    /** Комунікатор для спілкування зі службою. */
    Messenger mService = null;

    /** Прапорець, який вказує, чи ми визвали прив’язку на службу.
*/
    boolean mBound;

    /**
     * Клас для взаємодії з основним інтерфейсом сервісу.
     */
    private ServiceConnection mConnection = new
ServiceConnection() {
        public void onServiceConnected(ComponentName className,
IBinder service) {
            // Це визивається, коли з’єднання з сервісом було
            // встановлено, що дає нам об’єкт, ми можемо викори-
стовувати
 // для взаємодіяти зі службою. Ми спілкуємося зі службою за
            // допомогою Messenger, тому тут ми отримуємо на
       // стороні клієнта уявлення про це з вихідного об’єкта
             // IBinder.
            mService = new Messenger(service);
            mBound = true;
        }

        public void onServiceDisconnected(ComponentName className)
{
            // Це визивається, коли з’єднання з сервісом було
            // несподівано вимикається - тобто, процес його впав.
            mService = null;
            mBound = false;
        }
    };

    public void sayHello(View v) {
        if (!mBound) return;
        // Створити і відправити повідомлення в службу, використо-
вуючи 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);
        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;
        }
    }
}
```

Зверніть увагу, що в цьому прикладі не показано, як служба відповідає клієнту. Якщо потрібно, щоб служба реагувала, необхідно створити об’єкт `Messenger` і в клієнті. Потім, коли клієнт отримує зворотний виклик `onServiceConnected()`, вона відправляє в службу об’єкт `Message`, який включає об’єкт `Messenger` клієнта як значення параметра `replyTo` методу `send()`.

Приклад організації двостороннього обміну повідомленнями наведено в прикладах коду `MessengerService.java` (служба) і `MessengerServiceActivities.java` (клієнт).

Порівняння з AIDL. Коли необхідно організувати взаємодію між процесами, використання об'єкта `Messenger` для інтерфейсу набагато простіше від реалізації за допомогою AIDL, оскільки об'єкт `Messenger` поміщає в чергу всі запити до служби, тоді як інтерфейс, оснований виключно на AIDL, відправляє службі кілька запитів одночасно, а для цього потрібна підтримка багатопотоковості.

У більшості додатків служба не повинна підтримувати багатопотоковість, тому при використанні об'єкта `Messenger` служба може одночасно обробляти один запит. Якщо вам важливо, щоб служба була багатопотоковою, то для визначення інтерфейсу слід використовувати `AIDL`.

3.9.5. Прив'язка до служби

Для прив'язки 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'язка 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()`. Интерфейс `ServiceConnection` включає метод зворотного виклику, який система використовує для того, щоб видати об'єкт `IBinder`.

Примітка. Виконати прив'язку до служби можуть тільки операції, інші служби і постачальники контенту – **не можна** самостійно виконати прив'язку до служби з ресивера.

Тому для прив'язки до служби з клієнта необхідно виконати такі дії.

1. Реалізувати інтерфейс `ServiceConnection`. Ваша реалізація повинна перевизначати два методи зворотного виклику:

- `onServiceConnected()`. Система викликає цей метод, щоб видати об'єкт `IBinder`, повернутий методом `onBind()` служби.
- `onServiceDisconnected()`. Система Android викликає цей метод в разі непередбаченої втрати з'єднання зі службою, наприклад, при збої в роботі

служби або в разі її завершення. Цей метод не викликається, коли клієнт скасовує прив'язку.

2. Викликати метод `bindService()`, передавши в нього реалізацію інтерфейсу.

3. Коли система викликає ваш метод зворотного виклику `onServiceConnected()`, можна приступити до виконання викликів до служби за допомогою методів, визначених інтерфейсом.

4. Щоб відключитися від служби, треба викликати метод `unbindService()`.

У разі знищення клієнта виконується скасування його прив'язки до служби, проте вам завжди слід відміняти прив'язку по завершенні взаємодії зі службою або в разі припинення операції, щоб служба могла завершити свою роботу, коли вона не використовується.

Нижче наведено приклад фрагмента коду для підключення клієнта до створеної вище службі шляхом розширення класу `Binder` – клієнту потрібно лише передати повернутий об'єкт `IBinder` у клас `LocalService` і запросити екземпляр `LocalService`:

```
LocalService mService;
private ServiceConnection mConnection = new ServiceConnection() {
    // Викликається, коли з'єднання з сервісом встановлено
    public void onServiceConnected(ComponentName className, IBinder
service) {
        // Тому що ми зобов'язані явному сервісу, який працює
        // в нашому власному процесі, ми можемо відкинути його
IBinder
        // до конкретного класу і отримати доступ до нього.
        LocalBinder binder = (LocalBinder) service;
        mService = binder.getService();
        mBound = true;
    }

    // Викликається, коли з'єднання з сервісом роз'єднується неспо-
дівано
    public void onServiceDisconnected(ComponentName className) {
        Log.e(TAG, "onServiceDisconnected");
        mBound = false;
```

```
    }
};
```

За допомогою інтерфейсу `ServiceConnection` клієнт може виконати прив'язку до служби, передавши її в методі `bindService()`.

Наприклад:

```
Intent intent = new Intent(this, LocalService.class);
bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
```

Перший параметр в методі `bindService()` являє собою об'єкт `Intent`, який явно іменує службу для прив'язки (хоча перехід може бути і неявним).

Другий параметр – це об'єкт `ServiceConnection`.

Третій параметр являє собою прапорець, який вказує параметри прив'язки. Зазвичай ним є `BIND_AUTO_CREATE`, який створює служба, якщо вона вже не виконується. Інші можливі значення: `BIND_DEBUG_UNBIND` і `BIND_NOT_FOREGROUND` або 0, якщо значення відсутнє.

3.9.6. Управління життєвим циклом прив'язаної служби

Коли виконується скасування прив'язки служби до всіх клієнтів, система Android знищує таку службу (якщо вона не була запущена разом з `onStartCommand()`). У такому випадку не потрібно управляти життєвим циклом своєї служби, якщо вона виключно прив'язана служба – система Android управляє нею за вас на підставі прив'язки служби до будь-яких інших клієнтів.

Однак ви якщо вирішите реалізувати метод зворотного виклику `onStartCommand()`, то необхідно явно зупинити службу, оскільки в цьому випадку вона вважається запущеною. В такому випадку служба виконується доти, поки сама не зупинить свою роботу за допомогою методу `stopSelf()` або доти, поки інший компонент не викличе метод `stopService()` незалежно від прив'язки служби до будь-яких клієнтів.

Крім того, якщо ваша служба запущена і приймає прив'язку, то при виклику системою вашого методу `onUnbind()` можна повернути `true`, якщо ви бажаєте отримати виклик до `onRebind()` при наступній прив'язці до служби (замість отримання виклику до методу `onBind()`). Метод `onRebind()` повертає значення void, однак клієнт, як і раніше, отримує об'єкт `IBinder` в своєму методі зворотного виклику `onServiceConnected()`. На рис. 3.2 проілюстрована логіка життєвого циклу такого роду.

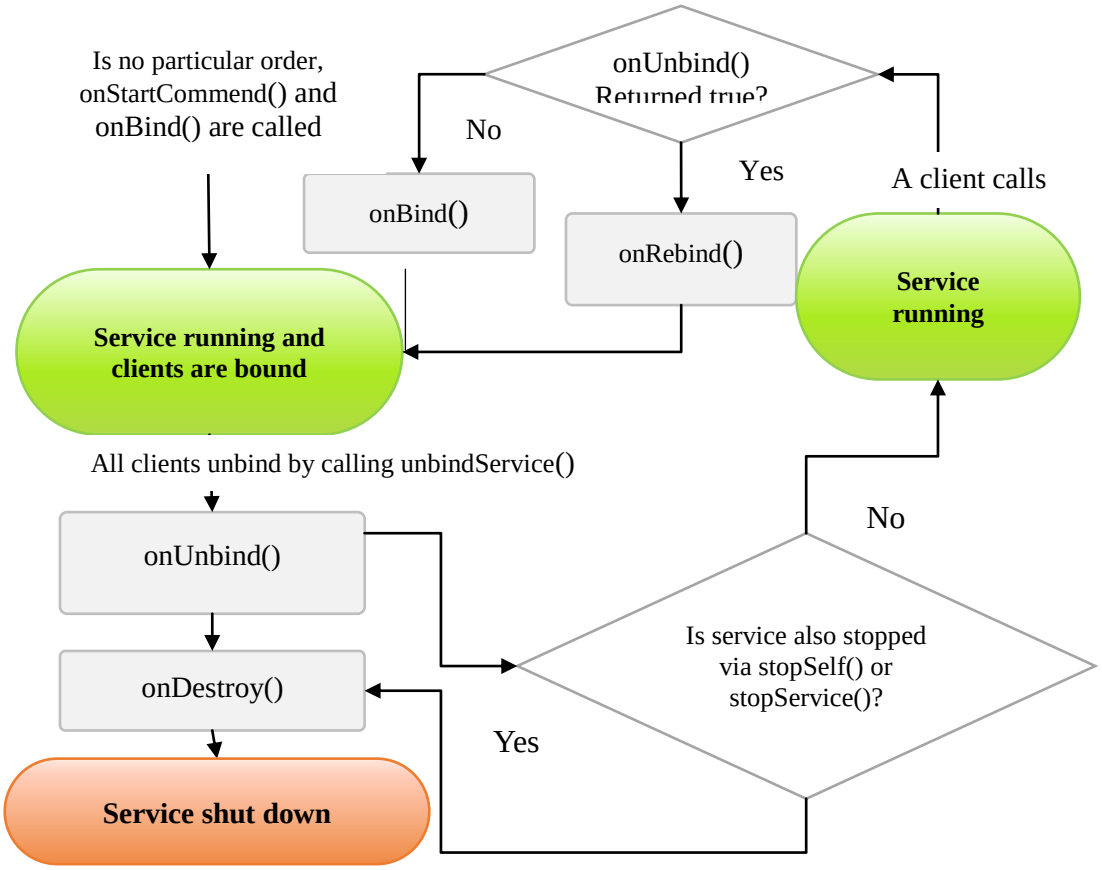

Рисунок 3.2 – Життєвий цикл запущеної служби, для якої виконується прив’язка

Запитання для самоконтролю

1. У чому полягає призначення служб?
2. Наведіть життєвий цикл служби.
3. Опишіть процес створення служби.
4. Як запускати службу на передньому плани процесів?
5. Опішить процес прив’язки до служби.

4. ЗБЕРЕЖЕННЯ ТА ОБРОБКА ДАНИХ В МОБІЛЬНИХ ДОДАТКАХ

Постачальники контенту (Content providers) керують доступом до структурованого набору даних. Вони інкапсулюють дані і надають механізми забезпечення їх безпеки. Постачальники контенту – це стандартний інтерфейс для об'єднання даних в одному процесі з кодом, який виконується в іншому процесі [38–40].

Коли вам потрібен доступ до даних у постачальнику контенту, використовуйте об'єкт `ContentResolver` в інтерфейсі `Context` вашого програмного додатка, щоб підключитися до постачальника як клієнт. Об'єкт `ContentResolver` 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 і повертає результати.

Вам не потрібно розробляти власний постачальник, якщо ви не плануєте надавати доступ до своїх даних іншим програмам. Однак вам знадобиться власний постачальник для надання настроюваних пошукових підказок у вашому власному додатку. Вам також знадобиться власний постачальник, якщо необхідно копіювати і вставляти складні дані або файли зі свого програмного додатка в інші.

До складу системи Android входять постачальники контенту, які керують такими даними, як аудіо, відео, зображення і особиста контактна інформація. Деякі з постачальників вказані в довідковій документації для пакета `android.provider`. Працювати з цими постачальниками може будь-який додаток Android (проте з деякими обмеженнями) [38–40].

4.1. Основні відомості про постачальника контенту

Постачальник контенту управляє доступом до центрального сховища даних. Постачальник є компонентом програми Android, який часто має власний користувацький інтерфейс для роботи з даними. Однак постачальники контенту призначені в першу чергу для використання іншими програмними додатками, які отримують доступ до постачальника за допомогою клієнтського об'єкта постачальника. Разом постачальники і клієнти постачальників забезпечують узгоджений, стандартний інтерфейс доступу до даних, який також обробляє взаємодію між процесами і забезпечує захищений доступ до даних.

4.1.1. Огляд

Постачальник контенту надає дані зовнішнім програмним додаткам у вигляді однієї або декількох таблиць, аналогічних таблицям в реляційній базі да-

них. Рядок являє собою екземпляр деякого типу даних, що збираються постачальником, а кожен стовпець в цьому рядку – це окремий елемент даних, зібраних для екземпляра.

Прикладом вбудованого постачальника в платформі Android може служити користувацький словник, в якому зберігаються дані про написання нестандартних слів, доданих користувачем. У табл. 4.1 показано, як дані можуть виглядати в цій таблиці постачальника.

Таблиця 4.1 – Приклад таблиці словника

word	app id	frequency	locale	_ID
mapreduce	user1	100	en_US	1
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 цієї таблиці наведено екземпляр слова, яке відсутнє в стандартному словнику. У кожному її стовпці містяться деякі дані для слова, наприклад дані про мову, в якої це слово було вперше використано. Заголовки стовпців – це імена стовпців, які зберігаються в постачальнику. Щоб дізнатися мову рядка, необхідно звернутися до стовпця locale. У цьому постачальнику стовпець _ID виступає в ролі «основного ключа», який постачальник автоматично зберігає.

Примітка. У постачальника необов’язково повинен бути основний ключ, а також йому необов’язково використовувати `_ID` як ім’я стовпця основного ключа, якщо такий є. Однак, якщо необхідно прив’язати дані з постачальника до класу `ListView`, один із стовпців повинен іменуватися `_ID`.

4.1.2. Доступ до постачальника

Для доступу програми до даних з постачальника контенту використовується клієнтський об’єкт `ContentResolver`. В цьому об’єкті є методи, які викликають ідентичні методи в об’єкті постачальника, який є екземпляром одного з конкретних підкласів класу `ContentProvider`. У методах `ContentResolver` наведено основні функції CRUD (абревіатура create, retrieve, update, delete [створення, отримання, оновлення та видалення]) постійного сховища.

Об’єкт `ContentResolver` у процесі клієнтської програми і об’єкт `ContentProvider` у програмі, яка володіє постачальником, автоматично об-

роблюють взаємодію між процесами. Об'єкт `ContentProvider` також виступає в ролі рівня абстракції між репозиторієм даних і зовнішнім поданням даних у вигляді таблиць.

Примітка. Для доступу до постачальника ваша програма зазвичай має запросити певні дозволи в своєму файлі маніфесту.

Наприклад, щоб отримати з постачальника користувальницького словника список слів і мов, на яких вони наведені, викличте метод `ContentResolver.query()`. У свою чергу, метод `query()` викликає метод `ContentProvider.query()`, визначений постачальником словника. Далі у прикладі коду показаний виклик методу `ContentResolver.query()`:

```
// Запит до словника користувача і 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 виводу
рядків
```

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 у кожний отриманий рядок
selection	WHERE *col = value*	selection задає критерії для вибору рядків
selectionArgs	(Точний еквівалент відсутній. Заповнювачі ? замінюються аргументами выбору)	
sortOrder	ORDER BY *col,col,...*	sortOrder задає порядок відображення рядків в об`єкті Cursor

4.1.3. URI контенту

URI контенту – це URI, який визначає дані в постачальнику. URI контенту можуть включати символічне ім'я всього постачальника (його **центр**) і ім'я, яке вказує на таблицю (**шлях**). При виклику клієнтського методу для доступу до таблиці в постачальнику URI контенту цієї таблиці виступає в ролі одного з аргументів цього методу.

Константа `CONTENT_URI` в попередніх рядках коду містить URI контенту таблиці words в користувацькому словнику. Об'єкт `ContentResolver` аналізує центр URI і використовує його для «вирішення» постачальника шляхом порівняння центру з системною таблицею відомих постачальників. `ContentResolver` може відправити аргументи запиту до відповідного постачальника.

`ContentProvider` використовує частину URI контенту, в якій вказано шлях, для вибору таблиці для доступу. У постачальника зазвичай є шлях для кожної наданої їм таблиці.

У попередніх рядках коду повний URI для таблиці words виглядає таким чином:

```
content://user_dictionary/words
```

Рядок `user_dictionary` – це центр постачальника, а рядок `words` – це шлях до таблиці. Рядок `content://` (**схема**) присутній 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, в стовпці `_ID` якого задано 4, можна скористатися таким URI контенту:

```
Uri singleUri =
ContentUris.withAppendedId(UserDictionary.Words.CONTENT_URI,4);
```

Ідентифікатори часто використовуються у випадку, коли ви витягли набір рядків і оновлюєте або видаляєте один з 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, наведеного вище, для додавання ідентифікатора до URI контенту `UserDictionary` використовується метод `withAppendedId()`.

4.1.4. Отримання даних від постачальника

У цьому підрозділі розглядається порядок отримання даних від постачальника на прикладі постачальника користувацького словника.

Для повної ясності в прикладах коду, наведених в цьому розділі, методи `ContentResolver.query()` викликаються в потоці призначеного для користувача інтерфейсу. У реальному коді запити слід виконувати асинхронно в окремому потоці. Одним із способів реалізувати це є використання класу `CursorLoader.`

Щоб отримати дані з постачальника, виконайте такі основні дії.

1. Запитайте у постачальника дозвіл на читання.
2. Визначте код, який відповідає за відправку 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 вам необхідно вказати в маніфесті програми, що вам потрібен такий дозвіл. Для цього скористайтеся елементом `<uses-permission>` і вкажіть точну назву дозволу, зазначену постачальником. Вказавши цей елемент в маніфесті, ви тим самим запитуєте необхідний дозвіл для вашої програми. Коли користувачі встановлюють вашу програму, вони отримують дозвіл на цей запит.

Щоб дізнатися точну назву дозволу на читання в використовуваному постачальнику, а також назви інших використовуваних в ньому дозволів на читання, зверніться до документації постачальника.

Постачальник користувацького словника задає дозвіл `android.permission.READ_USER_DICTIONARY` в своєму файлі маніфесту, тому програмі, якій потрібно виконати читання даних з постачальника, необхідно запросити саме цей дозвіл.

Створення запиту. Наступним етапом отримання даних від постачальника є створення запиту. У наступному фрагменті коду задаються деякі змінні для доступу до постачальника словника:

```
// projection – це масив стовпців, які необхідно включити у кож-
ний отриманий 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 містіть аргументі для вибірки
String[] mSelectionArgs = {""};
```

У наступному фрагменті коду демонструється 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 і порядок сортування.

Набір стовпців, які повинен повернути запит, називається **проекцією** (змінна `mProjection`).

Вираз, який задає рядки для отримання, складається з пропозиції вибору і аргументів вибору. Пропозиція вибору являє собою поєднання логічних виразів, імен стовпців і значень (змінна `mSelectionClause`). Якщо замість значення вказати параметр, що підставляється, метод запиту витягує значення з масиву аргументів вибору (змінна `mSelectionArgs`).

У наступному фрагменті коду, якщо користувач не вказав слово, то для пропозиції вибору задається значення null, а запит повертає всі слова, наявні в постачальника. Якщо користувач вказав слово, то для пропозиції вибору задається значення `UserDictionary.Words.WORD + "=?"`, а для першого елемента в масиві аргументів вибору – введене користувачем слово.

```
 * 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)) {
    // Встановлення значення null для змінної вибірки, таким чином ре-
зультатом будуть усі можливі слова
    mSelectionClause = null;
    mSelectionArgs[0] = "";

} else {
    // Встановлює умови вибору, що відповідає слову, що ввів користу-
вач.
    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 виводу рядків

// Деякі провайдери повертають нульове значення, якщо відбувається по-
милка, інші кидають виняткову ситуацію
if (null == mCursor) {
    /*
     * Вставте тут код для обробки виняткової ситуації. Не використо-
вуйте курсор! Ви можете викликати android.util.Log.e, щоб записати ін-
формацію про цю виняткову ситуацію.
     */
// Якщо курсор пустий, то провайдер не знаходить співпадінь
} else if (mCursor.getCount() < 1) {

    /*
     * Вставте тут код, щоб повідомити користувача, що пошук не викона-
вся успішно. Це не обов’язково.
     */

} else {
    // Вставте код тут, щоб обробити результати
}
```

Цей запит аналогічний SQL-інструкції:

```
SELECT _ID, word, locale FROM words WHERE word = <userinput>
ORDER BY word ASC;
```

У цій інструкції SQL замість констант класу-контракту використовуються фактичні імена стовпців.

Захист від введення 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 вставлення коду SQL.

Розглянемо наступну умову на вибірку:

```
// Складаємо умову на вибірку шляхом складання змінної, що ввів
користувач та назви стовпця
String mSelectionClause =  "var = " + mUserInput;
```

При використанні цієї умови, ви дозволяєте користувачеві зв’язати вашу інструкцію SQL з шкідливим кодом SQL. Наприклад, користувач може ввести

nothing; DROP TABLE *; для `mUserInput`, що призведе до створення наступної умови вибору: `var = nothing; DROP TABLE *;`. Оскільки умова вибору виконується в потоці як інструкція SQL, це може призвести до того, що постачальник видалить всі таблиці у відповідній базі даних SQLite (якщо тільки постачальник не налаштований на відстеження спроб вставити шкідливий код SQL).

Щоб уникнути цього, скористайтеся умовою вибору «?» виступає як параметр, що підставляється, або масивом аргументів вибору. Після цього введення даних користувачем буде пов'язане безпосередньо із запитом і не буде інтерпретуватися як частина інструкції SQL. Оскільки в цьому випадку запит, що ввів користувач, не розглядається як код SQL, то в нього не вдасться впровадити шкідливий код SQL. Замість об'єднання, яке варто включити в користувацький ввід даних, використовуйте наступну умову вибору:

```
// Створює пункт вибору зі змінним параметром
String mSelectionClause =  "var = ?";
```

Налаштуйте масив аргументів вибору таким чином:

```
// Defines an array to contain the selection arguments
String[] selectionArgs = {""};
```

Вкажіть значення для масиву аргументів вибору:

```
// Sets the selection argument to the user's input
selectionArgs[0] = mUserInput;
```

Умова вибору, в якій «?» використовується як параметр, що підставляється, 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 в проекції запиту для рядків, які відповідають критеріям вибірки в запиті. Об'єкт `Cursor` надає прямий доступ на читання рядків і стовпців, що містяться в ньому. За допомогою методів `Cursor` можна виконати ітерацію по рядках в результатах, визначити тип даних для кожного стовпця, отримати дані з шпальти, а також перевірити інші властивості результатів. Деякі реалізації об'єкта `Cursor` автоматично оновлюють об'єкт при зміні даних в постачальнику або запускають виконання методів в об'єкті-спостерігачі при зміні 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`, в якому для методу `Cursor.getCount()` вказано значення «0» (порожній об'єкт `Cursor`).

При виникненні внутрішньої помилки результати запиту залежать від певного постачальника. Постачальник може повернути `null` або видати `Exception`.

Оскільки `Cursor` являє собою «список» рядків, то найкращим способом відобразити вміст об'єкта `Cursor` буде зв'язати його з `ListView` за допомогою `SimpleCursorAdapter.`

Наступний фрагмент коду є продовженням попереднього фрагмента. Він створює об'єкт SimpleCursorAdapter, що містить объект `Cursor`, який був отриманий в запиті, а потім визначає цей об'єкт як адаптер для `ListView`:

```
// Визначає список стовпців для отримання з курсора і завантаження у рядок виводу
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 в XML для одного рядка в ListView
    mCursor,                               // Результат запиту
    mWordListColumns,                      // Масив імен колонок в курсорі
    mWordListItems,                        // Масив view IDs
   0);                                     // Прапори (зазвичай нема в них необхідності)

// Встановлюється адаптер для the ListView
mWordList.setAdapter(mCursorAdapter);
```

Примітка. Щоб повернути ListView з об'єктом Cursor, об'єкт cursor повинен містити стовпець з ім'ям _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 з результатів запиту. Замість того, щоб просто переглянути результати запиту, можна використовувати їх для виконання інших завдань. Наприклад, можна отримати написання слів з словника користувача, а потім виконати їх пошук в інших постачальників. Для цього виконайте ітерацію по рядках в об'єкті Cursor:

```
// 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 винтякову  ситуацію (помилку)

     */
    while (mCursor.moveToNext()) {

        // Отримуємо значення стовпця.
        newWord = mCursor.getString(index);

        // Вставте тут код для обробки отриманого слова
        ...

        // кінець циклу    }
} else {

    // Вставте тут код для звіту про помилку, якщо курсор є null або провайдер викинув виняткову ситуацію.
}
```

Реалізації об'єкта Cursor містять кілька методів get для отримання об'єкта різних типів даних. Наприклад, в наступному фрагменті коду

використовується метод `getString()`. У них також є метод `getType()`, який повертає значення, яке вказує на тип даних стовпця.

4.1.5. 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 додаток буде намагатися отримати доступ. На основі вимог постачальника інші програми запитують дозволи, які потрібні їм для доступу до постачальника. Кінцеві користувачі бачать запитані дозволи при установленні програми.

Якщо додаток постачальника не задає ніяких дозволів, інші додатки не отримують доступу до даних постачальника. Однак компонентам додатка постачальника завжди надано повний доступ на читання і запис, незалежно від заданих дозволів.

Як вже було зазначено раніше, для отримання даних з постачальника користувацького словника потрібен дозвіл `android.permission.READ_USER_DICTIONARY`. У постачальника передбачений окремий дозвіл `android.permission.WRITE_USER_DICTIONARY` для вставки, оновлення або видалення даних. Щоб отримати необхідні дозволи для доступу до інтернету, програма запитує їх з допомогою елемента `<uses-permission>` у файлі маніфесту. При установленні менеджером пакетів Android програми користувачеві необхідно затвердити всі дозволи, що необхідні додатку. У разі затвердження всіх дозволів менеджер пакетів продовжує установлення; якщо ж користувач відхиляє їх, менеджер пакетів скасовує установлення.

Для запиту доступу на читання даних у постачальника користувацького словника використовується наступний елемент `<uses-permission>`:

```
<uses-permission android:name="android.permission.READ_USER_DICTIONARY">
```

4.1.6. Вставка, оновлення та видалення даних

Подібно до того, як отримуються дані від постачальника, також можна використовувати можливості взаємодії між клієнтом постачальника і об’єктом `ContentProvider` постачальника для зміни даних. Можна викликати метод об’єкта `ContentResolver`, вказавши аргументи, які були передані у відповідний метод об’єкта `ContentProvider`. Постачальник і клієнт постачальника автоматично обробляють взаємодію між процесами і забезпечують безпеку.

Вставка даних. Для вставки даних у постачальник викличте метод `ContentResolver.insert()`. Цей метод вставляє новий рядок у постачальник і повертає URI контенту для цього рядка. У наступному фрагменті коду демонструється порядок вставки нового слова у постачальник користувацького словника:

```
// Визначаємо новий об'єкт Uri, який отримує результат вставки
Uri mNewUri;
// Визначаємо об'єкт, який містить нові значення для вставки
ContentValues mNewValues = new ContentValues();
/*
 * Встановлюємо значення кожного стовпця і 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 вставки
);
```

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 повинні містити дані такого ж типу, і якщо ви не збираєтеся вказувати значення, то можна задати для стовпця значення `null` за допомогою методу `ContentValues.putNull()`.

Код в наведеному фрагменті не додає стовпець `_ID`, оскільки цей стовпець зберігається автоматично. Постачальник присвоює унікальне значення `_ID` кожному доданому рядку. Зазвичай постачальники використовують це значення як основний ключ таблиці.

URI контенту, повернений в елементі `newUri`, служить для ідентифікації нового доданого рядка в наступному форматі:

```
content://user_dictionary/words/<id_value>
```

`<id_value>` – це вміст стовпця `_ID` для нового рядка. Більшість постачальників автоматично визначають цю форму URI контенту, а потім виконують необхідну операцію з необхідним рядком.

Щоб отримати значення `_ID` з повернутого об'єкта `Uri`, викличте метод `ContentUris.parseId()`.

Оновлення даних. Щоб оновити рядок, використовуйте об'єкт `ContentValues` з оновленими значеннями (так само, як ви це робите при вставці) та критеріями вибірки (так само, як і з запитом). Використовуваний вами клієнтський метод називається `ContentResolver.update()`. Вам не потрібно додавати значення в об'єкт `ContentValues` для оновлюваних стовпців. Щоб очистити вміст стовпця, встановіть значення `null`.

Наступний фрагмент коду служить для зміни мови у всіх рядках, де у якості мови зазначено en, на значення null. Обчислене значення – це кількість рядків, які були оновлені.

Також слід перевірити введені користувачем дані при 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;

/*
 * Встановлюємо обновлене значення і обновлюємо обрані слова.
 */
mUpdateValues.putNull(UserDictionary.Words.LOCALE);

mRowsUpdated = getContentResolver().update(
    UserDictionary.Words.CONTENT_URI,   // URI контенту словника
користувача
    mUpdateValues                       // колонка для обновлення
даних
    mSelectionClause                    // стовпець, по якому ви-
бирають
    mSelectionArgs                      // значення для порівнян-
ня
);
```

Видалення даних. Видалення даних аналогічно одержанню даних рядка: необхідно вказати критерії вибірки для рядків, які потрібно видалити, після чого клієнтський метод поверне кількість видалених рядків. Нижче наведено фрагмент коду для видалення рядків з ідентифікатором `appid user`. Метод повертає кількість видалених рядків.

Також слід перевірити дані, введені користувачем при виклику методу `ContentResolver.delete()`.

```
// Визначаємо критерії вибору для рядків, які ви хочете видалити
     String mSelectionClause = UserDictionary.Words.APP_ID + "
LIKE ?";
     String[] mSelectionArgs = {"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
користувача
    mSelectionClause                    // стовпець, по якому ви-
бирають
    mSelectionArgs                      // значення для
порівняння
);.
```

4.1.7. Типи даних постачальників

Постачальники контенту можуть надавати різні типи даних. Постачальник користувацького словника надає тільки текст, але він також може надавати такі формати:

– ціле число;
– довге ціле число (long);
– число з плаваючою комою;
– довге число з 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, зверніться до методів `get` класу `Cursor`.

Тип даних для кожного стовпця в постачальнику зазвичай вказується у документації до постачальника. Типи даних для постачальника користувацького словника вказані 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 ваш додаток обробляти пропоновані постачальником дані, а також для вибору типу обробки на основі типу MIME. Інформація про тип MIME зазвичай потрібна при роботі з постачальником, який

містить складні структури даних або файли. Наприклад, у таблиці `ContactsContract.Data` у постачальника контактів використовуються типи MIME для відзначення типу даних контакту, які зберігаються у кожному рядку. Щоб отримати тип MIME, відповідний до URI контенту, викличте метод `ContentResolver.getType()`.

Синтаксис стандартних і настроюваних типів MIME описаний у довідці за типами MIME.

4.1.8. Альтернативні форми доступу до постачальника

При розробці програми варто враховувати три альтернативні форми доступу до постачальника:

1. *Пакетний доступ*: можна створити пакет викликів доступу з використанням методів у класі `ContentProviderOperation`, а потім застосувати їх за допомогою методу `ContentResolver.applyBatch()`.

2. *Асинхронні запити*: запити слід виконувати в окремому потоці. Одним із способів реалізувати це є використання об'єкта `CursorLoader`.

3. *Доступ до даних за допомогою намірів*: незважаючи на те, що намір неможливо відправити безпосередньо в постачальник, можна відправити запит на додаток постачальника, в якому зазвичай є більше можливостей для зміни даних постачальника.

Пакетний доступ і зміна з допомогою намірів описані в наступних розділах.

Пакетний доступ. Пакетний доступ до постачальника корисно використовувати у випадках, коли необхідно вставити велику кількість рядків, або для вставки рядків в кілька таблиць в рамках 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 в постачальник контенту за допомогою методу `ContentResolver.applyBatch()`. В цей метод необхідно передати *центр* постачальника контенту, а не певний URI контенту. Це дозволить кожному об'єкту `ContentProviderOperation` в масиві взаємодіяти з різними таблицями. Метод `ContentResolver.applyBatch()` повертає масив результатів.

В описі класу-контракту ContactsContract.RawContacts також наведено фрагмент коду, в якому демонструється вставка в пакетному режимі. У вихід-

ному 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. Користувачам можна дозволити доступ до даних у постачальника, навіть у тому випадку, якщо у додатка відсутні дозволи на доступ, або шляхом отримання результуючого 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 у вас немає необхідних дозволів на доступ, шляхом надсилання наміру в додаток, у якого є такі дозволи, і отримання результуючого наміру, який містить дозволи URI. Ці дозволи для певного URI контенту діють доти, поки не буде завершена операція, яка отримала їх. Додаток, у якого є безстрокові дозволи, надає тимчасові дозволи шляхом встановлення відповідного прапорця в результуючому намірі:

– дозвіл на читання: FLAG_GRANT_READ_URI_PERMISSION;
– дозвіл на запис: FLAG_GRANT_WRITE_URI_PERMISSION.

Примітка. Ці прапорці не надають доступ на читання або запис постачальнику, центр якого вказаний в URI контенту. Доступ надається тільки самому URI.

Постачальник визначає дозволи URI для URI контенту у своєму маніфесті за допомогою атрибута android:grantUriPermission елемента <provider>, а також з допомогою дочірнього елемента <grant-uri-permission> елемента <provider>.

Наприклад, можна отримати дані про контакт з постачальника контактів, навіть якщо у вас немає дозволу READ_CONTACTS. Можливо, це потрібно реалізувати в додатку, який відправляє електронні привітання контакту в день його народження. Замість запиту READ_CONTACTS, коли ви отримуєте доступ до всіх контактів користувача та інформації про них, можна надати користувачеві можливість вказати, які контакти використовуються вашим додатком. Для цього скористайтеся наведеними нижче процесами.

1. Ваша 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 `setResult(resultcode, intent)` для створення наміру, який буде передано назад у вашу програму. Намір містить URI вмісту вибраного користувачем контакту, а також прапори `FLAG_GRANT_READ_URI_PERMISSION` додаткових даних. Ці прапори надають вашому додатку дозвіл URI на читання даних контакту, на який вказує URI контенту. Потім операція вибору викликає метод `finish()`, щоб повернути управління вашим додатком.

4. Ваша операція повертається на передній план, а система викликає метод `onActivityResult()` операції. Цей метод отримує результуючий намір, створений операцією вибору у програмі «Контакти».

5. За допомогою URI контенту з результуючого наміру можна виконати читання даних контакту з постачальника контактів, навіть якщо ви не просили у постачальника постійний доступ на читання в своєму маніфесті. Можна отримати інформацію про день народження контакту або відомості про його адресу електронної пошти, а потім відправити контакту електронне привітання.

Використання іншого додатку. Простий спосіб дозволити користувачеві змінювати дані, на доступ до яких у вас немає доступу – 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 призначений для користувача інтерфейс програми для вставки. Можна передати в цьому намірі додаткові дані, які додаток використовує для заповнення полів в інтерфейсі. Оскільки синтаксис повторюваних подій досить складний, то події слід вставляти 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 додатку. Якщо вашій програмі не надано дозволу, то, 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 користувацький інтерфейс.

Додаток, в який надсилається намір, не обов’язково має бути пов’язаний з постачальником. Наприклад, у постачальника контактів можна створити форму

контакту, а потім відправити намір ACTION_VIEW, що містить URI контенту для зображення контакту в засобі перегляду зображень.

4.1.9. Класи-контракти

Клас-контракт визначає константи, які забезпечують для додатків можливість працювати з URI контенту, іменами стовпців, операціями наміру та іншими функціями постачальника вмісту. Класи-контракти не включені у постачальника; розробнику постачальника слід визначити їх і зробити їх доступними для інших розробників. Багато хто з постачальників, що включені в платформу Android, містять відповідні класи-контракти в пакеті android.provider.

Наприклад, у постачальника користувацького календаря є клас-контракт UserDictionary, що містить константи URI контенту та імен стовпців. URI вмісту таблиці words визначено у константі UserDictionary.Words.CONTENT_URI. У класі UserDictionary.Words також є константи імен стовпців, які використовуються у фрагментах коду прикладу програми. Наприклад, проекцію запиту можна визначити таким чином:

```
String[] mProjection =
{
    UserDictionary.Words._ID,
    UserDictionary.Words.WORD,
    UserDictionary.Words.LOCALE
};
```

Іншим класом-контрактом є клас ContactsContract для постачальника контактів. Один з його підкласів, ContactsContract.Intents.Insert, являє собою клас-контракт, який містить константи для намірів і їх даних.

4.1.10. 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:

```
type/subtype
```

Наприклад, добре відомий тип MIME text/html має тип text і підтип html. Якщо постачальник повертає цей тип URI, це означає, що рядок запиту, в якому використовується цей URI, поверне текст з тегами HTML.

Рядки з налаштованим типом MIME, які також називаються типами MIME постачальника, мають більш складні значення типів і підтипів. Значення типу завжди таке:

```
vnd.android.cursor.dir
```

для декількох рядків, або

```
vnd.android.cursor.item
```

для одного рядка.

Підтип залежить від постачальника. Вбудовані постачальники Android зазвичай містять простий підтип. Наприклад, коли програма «Контакти» створює рядок для номера телефону, вона задає наступний тип MIME в цьому рядку:

```
vnd.android.cursor.item/phone_v2
```

Зверніть увагу, що значення підтипу просто `phone_v2.`

Розробники постачальників можуть створювати свої власні шаблони підтипів на основі центру і назв таблиць постачальника. Наприклад, розглянемо постачальник, який містить розклад руху поїздів. Центром постачальника є `com.example.trains`, в якому містяться таблиці Line1, Line2 і Line3. У відповідь на таку URI вміст:

```
content://com.example.trains/Line1
```

для таблиці Line1 постачальник повертає такий тип MIME:

```
vnd.android.cursor.dir/vnd.example.line1
```

У відповідь на такий URI вміст:

```
content://com.example.trains/Line2/5
```

для рядка 5 таблиці Line2 постачальник повертає наступний тип MIME:

```
vnd.android.cursor.item/vnd.example.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 конетенту керує доступом до центрального сховища даних. Реалізація постачальника включає один чи кілька класів у додатку Android, а

також елементи у файлі маніфесту. Один з класів реалізує підклас `ContentProvider`, який виступає в ролі інтерфейсу між постачальником і іншими додатками. Незважаючи на те, що постачальники контенту спочатку призначені для надання доступу до даних іншим програмам, у вашому додатку, безсумнівно, можуть міститися операції, які дозволяють запитувати і змінювати дані, керовані постачальником.

4.2.1. Підготовка до створення постачальника

Перш ніж приступити до створення постачальника, виконайте наведені нижче дії.

1. Вирішіть, чи потрібен взагалі вам постачальник контенту. Постачальник контенту потрібен у випадках, якщо ви хочете реалізувати в своєму додатку одну або кілька таких функцій:

1) надання складних даних або файлів інших програм;

2) надання користувачам можливості копіювати складні дані з вашого додатка в інші програми;

3) надання пошукових підказок, що налаштовуються, за допомогою платформи пошуку.

Вам *не потрібен* постачальник для роботи з базою даних SQLite, якщо ви плануєте використовувати її виключно у вашому додатку.

2. Якщо ви ще не прийняли остаточне рішення, ознайомтеся з розділом «Основні відомості про постачальника контенту», щоб дізнатися детальніше про постачальників контенту.

Після цього можна приступати до створення постачальника. Для цього виконайте наведені нижче дії.

1. Спроектуйте базове сховище для своїх даних. Постачальник контенту може надати дані таким чином:

– *Дані для файлів.* Дані, які зазвичай надходять у файли, такі, як фотографії, аудіо чи відео. Файли слід зберігати в закритому просторі вашого додатка. У відповідь на запит файлу з іншої програми ваш постачальник може запропонувати дескриптор файлу.

– *Структуровані дані.* Дані, які зазвичай надходять в базу даних, масив або аналогічну структуру, слід зберігати в тій формі, яка сумісна з таблицями з рядків і стовпців. Рядок являє собою об’єкт, наприклад, користувача, позицію або облікову одиницю, стовпець – деякі дані про об’єкт, наприклад, ім’я користувача або вартість одиниці. Зазвичай дані такого типу зберігаються в базі даних SQLite, однак можна використовувати постійне сховище будь-якого типу.

2. Визначте конкретну реалізацію класу `ContentProvider` і його необхідні методи. Цей клас виступає в ролі інтерфейсу між вашими даними та іншою частиною системи Android.

3. Визначте рядок центру постачальника, його URI контенту і стовпців. Якщо треба, щоб додаток постачальника обробляв наміри, то необхідно визначити дії намірів, додаткові дані і прапорці. Крім того, необхідно визначити дозволи, які будуть необхідні програмі для доступу до ваших даних. Всі ці значення слід визначити як константи в окремому класі-контракті; надалі цей клас можна надати іншим розробникам.

4. Додайте інші додаткові компоненти, наприклад, демонстраційні дані або реалізацію адаптера `AbstractThreadedSyncAdapter`, який служить для синхронізації даних між постачальником і хмарним сховищем даних.

4.2.2. Проектування сховища даних

Постачальник контенту являє собою інтерфейс для передачі даних, збережених у структурованому форматі. Перш ніж створювати інтерфейс, визначте спосіб зберігання даних. Дані можна зберігати в будь-якій формі, а потім спроектувати інтерфейс для читання та запису даних при необхідності.

В Android є деякі технології зберігання даних:

1. 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` являє собою базовий клас для доступу до баз даних.

Зверніть увагу, що вам не обов'язково використовувати базу даних для реалізації свого репозиторія. Постачальник являє собою зовнішній набір таблиць, як у випадку з реляційною базою даних, однак це не є вимогою до внутрішньої реалізації постачальника.

2. Для зберігання файлів даних в Android передбачено різні API-інтерфейси для роботи з файлами. Якщо проектується постачальник, який пропонує мультимедійні дані, такі, як музика чи відео, можна створити постачальник, що об'єднує табличні дані і файли.

3. Для роботи з мережевими даними використовуйте класи `java.net` і `android.net`. Також ви можете синхронізувати мережеві дані з локальним сховищем даних (наприклад, з базою даних), а потім надати ці дані у вигляді таблиць або файлів. Такий тип синхронізації демонструється на прикладі програми адаптера синхронізації.

Рекомендації щодо проектування даних. Наведемо декілька порад і рекомендацій щодо проектування структури даних постачальника:

1. У табличних даних завжди повинен бути стовпець для «основного ключа», який постачальник зберігає у вигляді унікального числового значення для кожного рядка. Можете використовувати це значення для зв'язування рядка з рядками в інших таблицях (використовуючи його як «ключ»). Незважаючи на те, що можна використовувати будь-яке ім'я для цього стовпця, рекомендується вказати ім'я `BaseColumns._ID`, оскільки для зв'язування результатів запиту постачальника з `ListView` необхідно, щоб один з одержуваних стовпців називався `_ID`.

2. Якщо планується надавати растрові зображення або дуже великі фрагменти даних для файлів, то дані слід зберігати у файлах, а потім надавати їх окремо замість зберігання прямо в таблиці. У такому випадку вам необхідно повідомити користувачам вашого постачальника про те, що для доступу до даних їм потрібно скористатися методом ContentResolver.

3. Для зберігання даних різного розміру або з різною структурою використовуйте 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*. В таблиці такого типу визначаються стовпець основного ключа, стовпець типу MIME та один або кілька спільних стовпців BLOB. На зміст даних у стовпцях BLOB вказує значення у стовпці типу MIME. Завдяки цьому у тій самій таблиці можна зберігати рядки різних типів. Прикладом таблиці, що не залежить від схеми, може служити таблиця з даними постачальника контенту `ContactsContract.Data`.

4.2.3. Проектування URI контенту

URI контенту являє собою URI, який визначає дані у постачальника. URI контенту можуть включати символічне ім'я всього постачальника (його центр) і ім'я, яке вказує на таблицю або файл (шлях). Додаткова частина URI з ідентифікатором вказує на окремий рядок у таблиці. У кожного методу доступу до даних в 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 виступає як внутрішнє ім'я у системі Android. Щоб уникнути конфліктів з іншими постачальниками як основа центру постачальника повинні виступати відомості про володіння доменом в Інтернеті (в зворотному поряд-

ку). Оскільки ця рекомендація також застосовується і до назв пакетів Android, можна визначити центр свого постачальника у вигляді розширення назви пакета, в якому міститься постачальник. Наприклад, якщо пакет Android називається `com.example.<appname>`, то центром вашого постачальника має бути `com.example.<appname>.provider`.

Проектування структури шляху. Зазвичай розробники створюють URI контенту на основі центру постачальника, додаючи до нього шлях, який вказує на окремі таблиці. Наприклад, якщо є дві таблиці (*table1* і *table2*), центр постачальника з попереднього прикладу слід об'єднати для формування наступних URI контенту:

`com.example.<appname>.provider/table1`

і

`com.example.<appname>.provider/table2`.

Шляхи не обмежені одним сегментом, і не на кожному рівні шляху є таблиця.

Обробка ідентифікаторів URI контенту. Зазвичай постачальники надають доступ до одного рядка в таблиці шляхом прийняття URI контенту, в кінці якого вказано значення ідентифікатора рядка. Також постачальники зазвичай перевіряють збіг значення ідентифікатора по стовпцю `_ID` в таблиці і надають запитуваний доступ до відповідного рядка.

Це спрощує створення загального методу проектування для програм, які отримують доступ до постачальника. Додаток відправляє запит постачальнику і відображає отриманий в результаті такого запиту об'єкт `Cursor` в об'єкті `ListView` з 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 із об'єкта `Cursor` в базовому об'єкті `ListView`, отримує значення `_ID` для цього рядка, додає його до URI контенту, а потім відправляє постачальнику запит на доступ. Після чого постачальник може запросити або змінити рядок, обраний користувачем.

Шаблони URI контенту. Для вибору дії з вхідним даними URI контенту, в API постачальника є клас `UriMatcher`, в якому є шаблони URI контенту з цілочисельними значеннями. Такі цілочисельні значення, які відповідають певним шаблонам, можна використовувати в операторі `switch`, який вибирає відповідну дію для URI контенту.

Для визначення збігу URI контенту з шаблоном використовуються такі символи:

* – відповідність рядку будь-якої довжини з будь-якими припустимими символами;

– відповідність рядку будь-якої довжини з цифрами.

Як приклад для проектування, написання коду для обробки URI контенту рекомендується використовувати центр поставщика `com.example.app.provider`, який розпізнає наступні 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 URI контент, якщо до них додано ідентифікатор рядка (наприклад, `content://com.example.app.provider/table3/1` для рядка з ідентификатором1 в таблиці `table3`.

Можливе використання наступних 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/#:
```

Співпадає 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, наведеного нижче, показано, як працюють методи в класі `UriMatcher`. Цей код обробляє URI для всієї таблиці іншим чином, ніж для URI для 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 і його шлях з цілочисельним значенням. Метод `match()` повертає ціле значення для URI. Оператор

`switch` вибирає, що йому треба виконати: запит всієї таблиці або тільки окремого запису:

```
public class ExampleProvider extends ContentProvider {
...
    // Створюємо об’єкт  UriMatcher.
    private static final UriMatcher sUriMatcher;
...
    /*
     * Виклики функції addURI() знаходяться тут, для всіх моделей URI контен-
ту, що слід розпізнати постачальнику. В цьому фрагменті коду показаний виклик
цієї функції лише до таблиці 3     */
...
    /*
     * Встановлюємо цілочисельне значення рівне 1 для кількох рядків у таблиці
3 */
    sUriMatcher.addURI("com.example.app.provider", "table3", 1);

    /*
     * Встановлюємо код для одного рядку рівним 2. У цьому випадку
використовується  знак "#" знак. 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 і 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 знайдений, ви повинні виконати обробку помилок
}
        // місце, щоб здійснити запит
```

```
    }
```

Інший клас, `ContentUris`, надає зручні методи для роботи з частиною id URI контенту. Класи `Uri` і `Uri.Builder` містять зручні методи для синтаксичного аналізу існуючих об'єктів `Uri` і для створення нових.

4.2.4. Реалізація класу ContentProvider

Екземпляр класу `ContentProvider` управляє доступом до структурованого набору даних шляхом опрацювання запитів від інших додатків. У кінцевому рахунку, при всіх формах 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 рамках вашого власного конкретного підкласу. Всі наведені нижче методи, крім `onCreate()`, викликаються клієнтським додатком, який намагається отримати доступ до вашого постачальника контенту.

1. `query()`. Отримання даних від постачальника. Використовує аргументи для вибору таблиці для запиту, рядків і стовпців, які потрібно повернути, і зазначає порядок сортування результатів. Повертає дані у вигляді об'єкта `Cursor`.

2. `insert()`. Вставка рядка в ваш постачальник. Використовує аргументи для вибору кінцевої таблиці та отримання значень стовпця, які слід використовувати. Повертає URI контенту для нового вставленого рядка.

3. `update()`. Оновлення існуючих рядків у постачальника. Використовує аргументи для вибору таблиці або рядків для оновлення, а також для отримання оновлених значень стовпця. Повертає кількість оновлених рядків.

4. `delete()`. Видалення рядків з постачальника. Використовує аргументи для вибору таблиці або рядків для видалення. Повертає кількість видалених рядків.

5. `getType()`. Повернення типу MIME, що відповідає URI контенту.

6. `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.

Підпис цих методів аналогічний до підпису для ідентичних методів в об'єкті `ContentResolver`.

При реалізації цих методів слід враховувати такі моменти.

1. Всі ці методи, крім `onCreate()`, можна викликати відразу з декількох потоків, тому вони повинні бути реалізовані із збереженням потокобезпеки.

2. Уникайте занадто довгих операцій в методі `onCreate()`. Відкладіть виконання завдань ініціалізації доти, поки вони не будуть потрібні.

3. Незважаючи на те, що повинні реалізувати ці методи, код необов'язково повинен 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 методом query(). Метод `ContentProvider.query()` повинен повертати об'єкт `Cursor`, а при збої видавати `Exception`. Якщо як сховище даних використовується база даних SQLite, можна просто повернути об'єкт `Cursor`, який був повернутий одним з 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 обробки запиту сталася внутрішня помилка.

Якщо ви не використовуєте базу даних SQLite як сховище даних, зверніться до одного з конкретних підкласів об'єкта `Cursor`. Наприклад, клас `MatrixCursor` реалізує об'єкт `cursor`, в якому кожен рядок являє собою масив класу `Object`. За допомогою цього класу скористайтеся методом `addRow()`, щоб додати новий рядок.

Слід пам'ятати, що система Android повинна мати можливість взаємодіяти з `Exception` в межах процесу. Система Android дозволяє це робити для зазначених нижче винятків, які можуть бути корисні при обробці помилок запитів:

– `IllegalArgumentException` (цей виняток можна видати у разі, якщо постачальник одержує неправильний URI контенту);

– `NullPointerException`.

Реалізація методу insert(). Метод `insert()` додає новий рядок у відповідний рядок, використовуючи значення в аргументі `ContentValues`. Якщо в аргументі `ContentValues` відсутнє ім'я стовпця, можливо, знадобиться вказати для нього значення за замовчуванням (або в коді постачальника, або у схемі бази даних).

Цей метод повинен повертати URI контенту для нового рядка. Для цього додайте 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 з вашого сховища даних. Якщо для роботи з постачальником використовується адаптер синхронізації, розгляньте можливість позначки віддаленого рядка прапором delete замість остаточного видалення рядка. Адаптер синхронізації може перевірити наявність видалених рядків з прапором delete, щоб видалити їх з сервера перед видаленням їх з 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 даними методами.

Реалізація методу onCreate(). Система Android викликає метод `onCreate()` при запуску постачальника. У цьому методі слід виконувати тільки завдання ініціалізації, що швидко виконуються, а створення бази даних і завантаження даних 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 ви використовуєте базу даних SQLite, в методі `ContentProvider.onCreate()` можна створити новий об'єкт `SQLiteOpenHelper`, а потім створити таблиці SQL при першому відкритті бази даних. Щоб спростити це, при першому виклику методу `getWritableDatabase()` він автоматично викликає метод `SQLiteOpenHelper.onCreate()`.

В наведених нижче фрагментах коду ілюструється взаємодія між методом `ContentProvider.onCreate()` і методом `SQLiteOpenHelper.onCreate()`. У першому фрагменті коду наведена реалізація методу `ContentProvider.onCreate()`:

```
public class ExampleProvider extends ContentProvider

    /*
     * Визначаємо дескриптор для допоміжного об'єкта бази даних. Клас
MainDatabaseHelper визначається
* у наступному фрагменті.
    private MainDatabaseHelper mOpenHelper;
    // Визначаємо ім'я 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 додатку
       DBNAME,              // ім’я бази даних)
            null,                //використовуємо SQLite курсор за замовчуван-
ням
            1                    // номер версії
        );
        return true;
    }

    // Реалізуємо метод Insert постачальника
    public Cursor insert(Uri uri, ContentValues values) {
        // Необхідно вставити тут код, щоб визначити, які таблиці відкриті,
обробити помилки, і так далі

        ...

        /*
         * Отримуємо записи бази даних.
         */
        db = mOpenHelper.getWritableDatabase();
    }
}
```

У наступному фрагменті коду наведена реалізація методу `SQLiteOpenHelper.onCreate()`, включаючи допоміжний клас:

```
...
// Рядок, яка визначає інструкцію SQL для створення таблиці
private static final String SQL_CREATE_MAIN = "CREATE TABLE " +
    "main " +                       // Назва таблиці
    "(" +                           // Стовпці в таблиці
    " _ID INTEGER PRIMARY KEY, " +
    " WORD TEXT"
    " FREQUENCY INTEGER " +
    " LOCALE TEXT )";
...
/**
 *  Клас Helper, який фактично створює і управляє основним сховищем даних про-
вайдера.
 */
protected static final class MainDatabaseHelper extends SQLiteOpenHelper {

    /*
     * Створює екземпляр відкритого помічника для SQLite сховища даних поста-
чальника
     * Не робити створення бази даних та оновлення тут.
     */
    MainDatabaseHelper(Context context) {
        super(context, DBNAME, null, 1);
    }

    /*
     * Створює сховище даних. Це викликається при спробі постачальника, щоб
відкрити
```

```
     * сховище і SQLite повідомляє, що він не існує.
     */
    public void onCreate(SQLiteDatabase db) {

        // Створює основну таблицю
        db.execSQL(SQL_CREATE_MAIN);
    }
}
```

4.2.5. Реалізація типів MIME постачальника контенту

У класі `ContentProvider` передбачено два методи для повернення типів MIME:

1) `getType()` – один з необхідних методів, який потрібно реалізувати для кожного постачальника;

2) `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 виступати як шаблон, а не як певний URI; в цьому випадку необхідно повернути тип даних, пов'язаний з URI контенту, який відповідає шаблонам.

Для загальних типів даних, таких, як текст, HTML або JPEG, метод `getType()` повинен повертати стандартний тип MIME. Повний список стандартних типів наведено на веб-сайті IANA MIME Media Types.

Для URI контенту, які вказують на один або кілька рядків табличних даних, метод `getType()` повинен повертати тип MIME у форматі MIME постачальника, який є в системі Android:

Частина типу: `vnd.`

Частина підтипу:

1) якщо шаблон URI призначений для одного рядка:

`android.cursor.item/`;

2) якщо шаблон URI призначений для декількох рядків:

`android.cursor.dir/`.

Частина постачальника: `vnd.<name>.<type>.`

Вказуєте `<name>` й `<type>`. Значення `<name>` має бути унікальним глобально, а значення `<type>` – унікальним для відповідного шаблону URI. Як `<name>` рекомендується використовувати назву вашої компанії або частину назви пакета Android вашого додатку. Як `<type>` рекомендується використовувати рядок, який визначає пов'язану з URI таблицю.

Наприклад, якщо центр постачальника є `com.example.app.provider`,

який надає таблицю `table1`, то тип MIME для кількох рядків у таблиці `table1` буде таким:

```
vnd.android.cursor.dir/vnd.com.example.provider.table1
```

Для одного рядка в таблиці `table1` тип MIME буде таким:

```
vnd.android.cursor.item/vnd.com.example.provider.table1
```

Типи MIME для файлів. Якщо постачальник надає файли, необхідно реалізувати метод `getStreamTypes()`. Цей метод повертає масив `String` з типами MIME для файлів, що поветаються постачальником для заданого URI контенту. Запропоновані постачальником типи слід сортувати за допомогою аргументу фільтра типів MIME, щоб повертались тільки ті типи MIME, які необхідно обробити клієнту.

Наприклад, розглянемо постачальника, який надає фотографії у вигляді файлів у форматах `.jpg`, `.png` і `.gif`. Якщо додаток викликає метод `ContentResolver.getStreamTypes()` з рядком фільтра `image/*` (щось подібне до «зображення»), то метод `ContentProvider.getStreamTypes()` повинен повертати наступний масив:

```
{ "image/jpeg", "image/png", "image/gif"}
```

Якщо ж додатку потрібні тільки файли `.jpg`, то він викликає метод `ContentResolver.getStreamTypes()` з рядком фільтра `*\/jpeg`; метод `ContentProvider.getStreamTypes()` при цьому повинен повертати таке:

```
{"image/jpeg"}
```

Якщо постачальник не надає жоден з типів MIME, запитаних в рядку фільтра, то метод `getStreamTypes()` повинен повертати `null`.

4.2.6. Реалізація класу-контракту

Клас-контракт являє собою клас `public final`, в якому містяться визначення констант для URI, імен стовпців, типів MIME та інших метаданих постачальника. Клас встановлює контрактні відносини між постачальником і іншими додатками шляхом забезпечення прямого доступу до постачальника навіть у разі зміни фактичних значень URI, імен стовпців і т. д.

Клас-контракт також корисний для розробників тим, що в ньому містяться мнемонічні імена для його констант, завдяки чому знижується ризик

того, що розробники скористаються неправильними значеннями для імен стовпців або URI. Оскільки це клас, він може містити документацію Javadoc. Інтегровані середовища розробки, такі, як Eclipse, можуть автоматично заповнювати імена констант з класу-контракту і відображати Javadoc для констант.

У розробників немає доступу до файлу класу-контракту з вашого додатка, проте вони можуть статично скомпілювати клас-контракт в свій додаток з наданого вами файлу `.jar`.

Прикладом класу-контракту може служити клас `ContactsContract` і його вкладені класи.

4.2.7. Реалізація дозволів постачальника контенту

Нижче наведено короткий огляд основних моментів.

1. За замовчуванням файли з даними зберігаються у внутрішньому сховищі пристрою і доступні тільки вашому додатку і постачальнику.

2. Створювані вами бази даних `SQLiteDatabase` також доступні тільки вашому додатку і постачальнику.

3. Файли з даними, які зберігаються в зовнішньому сховищі, за замовчуванням є *загальнодоступними, і їх може зчитати будь-який користувач*. Вам не вдасться використовувати постачальника контенту для обмеження доступу до файлів, які зберігаються в зовнішньому сховищі, оскільки додатки можуть використовувати інші виклики API для їх читання або запису.

4. Виклики методу для відкриття або створення файлів або баз даних SQLite, що знаходяться у внутрішньому сховищі на вашому пристрої, потенційно можуть надати всім іншим додаткам доступ як на запис, так і на читання даних. Якщо використовується внутрішній файл або база даних як сховище постачальника, в якому виконати читання або запис даних може будь-який користувач, то дозволів, заданих в маніфесті постачальника, буде явно недостатньо для захисту ваших даних. За замовчуванням доступ до файлів і баз даних у внутрішньому сховищі є закритим, і вам не слід змінювати параметри доступу до сховища вашого постачальника.

Якщо необхідно використовувати дозволи постачальника контенту для управління доступом до даних, то зберігайте дані у внутрішніх файлах, в базах даних SQLite або в хмарі (наприклад, на віддаленому сервері), а доступ до файлів і баз даних має бути наданий тільки вашому додатку.

Реалізація дозволів. Будь-який додаток може виконувати читання даних в постачальнику або записувати їх, навіть якщо відповідні дані є закритими, оскільки за замовчуванням для постачальника не задані дозволи. Щоб змінити

ці настройки, задайте дозволи для постачальника у файлі маніфесту за допомогою атрибутів елемента <provider> або його дочірніх елементів. Можна задати дозволи, які застосовуються до всього постачальника або тільки до певних таблиць, або навіть тільки до певних записів або всього дерева.

Для завдання дозволів використовується один або кілька елементів <permission> у файлі маніфесту. Щоб дозволи були унікальними для постачальника, використовуйте області, аналогічні Java, для атрибута android:name. Наприклад, надайте дозволу на читання ім'я com.example.app.provider.permission.READ_PROVIDER.

Нижче перераховані області дозволів для постачальника, починаючи з дозволів, які застосовуються 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, задається за допомогою атрибута android:permission елемента <provider>.

Окремий дозвіл на читання / запис на рівні постачальника. Дозволи на читання і запис для всього постачальника задаються 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. Це дозвіл на читання, запис або читання / запис для URI контенту в постачальника. Кожен URI, що підлягає керуванню, задається за допомогою дочірнього елемента <path-permission> елемента <provider>. Для кожного зазначеного вище URI контенту можна задати дозвіл на читання / запис, тільки читання або тільки запис, або всі три дозволи. Дозволи на читання і 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 в своєму маніфесті. Якщо включені тимчасові дозволи, то єдиними додатками, яким потрібні «постійні» дозволи на роботу з постачальником, є ті, які безперервно отримують доступ до всіх ваших даних.

Розглянемо приклад з дозволами, які необхідно реалізувати для постачальника електронної пошти, та додатками, 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, задайте тимчасові дозволи для URI контенту фотографій. Спроектуйте ваш додаток для роботи з електронною поштою таким чином, щоб у випадках, коли користувач бажає відобразити фотографію, додаток відправляв намір, в 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 у засоба перегляду відсутній звичайний дозвіл на читання даних з постачальника.

Щоб включити тимчасові дозволи, задайте атрибут `android:grantUriPermissions` для елемента `<provider>`, або додайте один або кілька дочірніх елементів `<grant-uri-permission>` в ваш елемент `<provider>`. Якщо використовуються тимчасові дозволи, вам необхідно викликати метод `Context.revokeUriPermission()` кожен раз, коли ви здійснюєте віддалену підтримку URI контенту з постачальника, а URI контенту пов'язаний 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, які потрібні на рівні постачальника або на рівні шляху.

Якщо для прапора задано значення false, вам необхідно додати дочірні елементи `<grant-uri-permission>` в свій елемент `<provider>`. Кожен дочірній елемент задає URI контенту, для яких надається тимчасовий дозвіл.

Щоб делегувати додатку тимчасовий доступ, в намірі повинні бути вказані прапорці `FLAG_GRANT_READ_URI_PERMISSION` або `FLAG_GRANT_WRITE_URI_PERMISSION` , або обидва прапорці. Ці прапорці задаються за допомогою методу `setFlags()`.

Якщо атрибут `android:grantUriPermissions` відсутній, передбачається, що його значення `false`.

4.2.8. Елемент <provider>

Як і компоненти `Activity` й `Service`, підклас класу `ContentProvider` повинен бути визначений у файлі маніфесту програми за допомогою елемента `<provider>`. Нижче вказана інформація, яку система Android отримує з цього елемента.

Центр постачальника (android:authorities). Символічні імена, які ідентифікують весь постачальник в системі.

Назва класу постачальника (android:name). Клас, який реалізує клас `ContentProvider`.

Дозволи. Атрибути, які визначають дозволи, необхідні іншим додаткам для доступу до даних в постачальника:

– `android:grantUriPermssions`: 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.

Додаткові відомості про дозволи і відповідні атрибути подані в розділі «Реалізація дозволів постачальника контенту».

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 у тому ж самому процесі);

– `android:multiProcess`: прапорець, що дозволяє системі запускати постачальника в тому ж процесі, що і викликає клієнт;

– `android:process`: назва процесу, в якому запускається постачальник;

– `android:syncable`: прапорець, який вказує на те, що дані в постачальнику слід синхронізувати з даними на сервері.

Інформаційні атрибути. Додатковий значок і мітка для постачальника:

– `android:icon`: графічний ресурс, що містить значок для постачальника. Значок відображається поруч з міткою постачальника в списку додатків в розділі *Налаштування> Програми> Все*.

– `android:label`: інформаційна мітка з описом постачальника або його дані, або обидва описи. Мітка відображається в списку додатків в розділі «*Налаштування> Програми> Все»*.

4.2.9. Наміри і доступ до даних

Додатки можуть отримувати доступ до контенту за допомогою об'єктів `Intent`. Додаток при цьому не викликає будь-які методи класів `ContentResolver` чи `ContentProvider`. Замість цього він відправляє намір операції, що запускається, яка зазвичай є частиною власного додатка постачальника. 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 пропонується можливість змінити ці дані, перш ніж використовувати їх для зміни даних в постачальника.

Можливо, доступ за допомогою наміру потрібно використовувати для забезпечення цілісності даних. Для вставки, оновлення та видалення даних в постачальника може існувати суворо певний 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 небхідно надати розробникам можливість доступу за допомогою намірів, вам слід ретельно задокументувати таку функцію. Поясніть їм, чому доступ за допомогою намірів через призначений для користувача інтерфейс вашої програми набагато кращий від зміни даних за допомогою їх коду.

Обробка вхідного наміри для зміни даних постачальника нічим не відрізняється від обробки інших намірів.

4.3. Завантажувачі (Loaders)

Завантажувачі, які з'явилися в Android 3.0, спрощують асинхронне завантаження даних в операцію або фрагмент. Завантажувачі мають певні властивості, а саме, вони:

– є наявними для будь-яких операцій `Activity` і фрагментів `Fragment`;

– забезпечують асинхронне завантаження даних;

– відстежують джерело своїх даних і видають нові результати при зміні контенту;

– можуть автоматично перепідключатися до останнього курсора завантажувача при відтворенні після зміни конфігурації. Таким чином, їм не потрібно повторно запитувати свої дані.

Зведена інформація про API-інтерфейсі завантажувача. Є кілька класів і інтерфейсів, які можуть використовувати завантажувачі в додатку (табл. 4.3).

Таблиця 4.3 – Властивості завантажувачів

Клас / інтерфейс	**Опис**
LoaderManager	Абстрактний клас, що пов’язується з Activity чи Fragment для управління одним або декількома інтерфейсами Loader. Це дозволяє додатку керувати операціями, які виконуються досить довго разом з життєвим циклом Activity чи Fragment; найчастіше цей клас використовується з CursorLoader, однак для додатка можуть писати свої власні завантажувачі для роботи з іншими типами даних. Є тільки один клас LoaderManager на операцію або фрагмент. Однак у класу 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 використовується CursorLoader, але можна реалізувати і власний підклас. Коли завантажувачі активні, вони повинні відслідковувати джерело своїх даних і видавати нові результати при зміні 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. Він будується на AsyncTaskLoader для виконання запиту до курсора у фоновому потоці, щоб не блокувати призначений для користувача інтерфейс програми. Використання цього завантажувача – це найкращий спосіб асинхронного завантаження даних з ContentProvider замість виконання керованого запиту через платформу або API-інтерфейси операції

Наведені в табл. 4.3 класи та інтерфейси є найбільш важливими компонентами, за допомогою яких в додатку реалізується завантажувач. При створенні кожного завантажувача не потрібно використовувати всі ці компоненти, проте завжди слід вказувати посилання на `LoaderManager` для ініціалізації завантажувача і використовувати реалізацію класу `Loader`, наприклад `CursorLoader`.

Використання завантажувачів в додатку. У цьому розділі описується використання завантажувачів в 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 створюються нові завантажувачі і ведеться управління посиланнями на існуючі завантажувачі;
- спосіб відображення даних завантажувача, наприклад `SimpleCursorAdapter`;
- джерело даних, наприклад `ContentProvider`, коли використовується `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 `onCreate()` операції або в методі фрагмента `onActivityCreated()`. Робиться це в такий спосіб:

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

Метод `initLoader()` приймає такі параметри:

- унікальний ідентифікатор, що позначає завантажувач. В даному прикладі ідентифікатором є 0;
- необов'язкові аргументи, які передаються завантажувачу при побудові (в даному прикладі це `null`);
- реалізація `LoaderManager.LoaderCallbacks`, яка викликає клас LoaderManager для видачі подій завантажувача. В даному прикладі локальний клас реалізує інтерфейс `LoaderManager.LoaderCallbacks`, тому він передає посилання самому собі: `this`.

Виклик `initLoader()` забезпечує ініціалізацію завантажувача. Можливий один з двох результатів:

1. Якщо завантажувач, вказаний за допомогою ідентифікатора, вже існує, то буде повторно використаний завантажувач, створений останнім.

2. Якщо завантажувач, вказаний за допомогою ідентифікатора, не існує, то `initLoader()` викликає метод LoaderManager.`LoaderCallbacks` з `onCreateLoader()`. Саме тут реалізується код для створення екземпляра і повернення нового завантажувача.

У будь-якому випадку реалізація `LoaderManager.LoaderCallbacks` зв'язується з завантажувачем і буде викликатися при зміні стану завантажувача. Якщо в момент цього виклику компонент знаходиться в запущеному стані, це означає, що запитаний завантажувач вже існує і сформував свої дані. В цьому випадку система відразу ж викличе `onLoadFinished()` (під час `initLoader()`), будьте готові до цього.

Зверніть 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 завантажувати і завершується при необхідності, а також підтримує стан завантажувача і пов'язаного з ним контенту. А це означає, що ви будете рідко взаємодіяти з завантажувачами безпосередньо (проте приклад використання методів завантажувача для тонкої настройки його поведінки див. у зразку коду `LoaderThrottle`). Для втручання в процес завантаження при виникненні певних подій зазвичай використовуються методи `LoaderManager.LoaderCallbacks`.

Перезапуск завантажувача. При використанні методу `initLoader()`, як показано вище, задіюється існуючий завантажувач із зазначеним ідентифікатором – в разі його наявності. Якщо такого завантажувача немає, метод його створить. Однак іноді старі дані потрібно відкинути і почати все заново.

Для видалення старих даних використовується метод `restartLoader()`. Наприклад, ця реалізація методу `SearchView.OnQueryTextListener` перезапускає завантажувач, коли змінюється запит користувача. Завантажувач необхідно перезавантажити, для того щоб він міг використовувати змінений фільтр пошуку для виконання нового запиту:

```
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;
}
```

Використання зворотних викликів класу LoaderManager. `LoaderManager.LoaderCallbacks` являє собою інтерфейс зворотного виклику, який дозволяє клієнту взаємодіяти з класом `LoaderManager`.

Очікується, що завантажувачі, зокрема `CursorLoader`, зберігатимуть свої дані після їх зупинки. Це дозволяє додаткам зберігати свої дані в методах `onStop()` і `onStart()` операції або фрагмента, з тим щоб, коли користувач повернеться в додаток, йому не довелося чекати, поки дані завантажаться заново. Методи `LoaderManager.LoaderCallbacks` використовуються, щоб дізнатися, коли потрібно створити новий завантажувач, а також для того, щоб вказати додатку, коли прийшов час перестати використовувати дані завантажувача.

Інтерфейс `LoaderManager.LoaderCallbacks` використовує такі методи:

– `onCreateLoader()` – створює екземпляр і повертає новий клас Loader для даного ідентифікатора;

– `onLoadFinished`() – викликається, коли створений раніше завантажувач завершив завантаження;

– `onLoaderReset`() – викликається, коли стан створеного раніше завантажувача скидається, в результаті чого його дані губляться.

Метод onCreateLoader. При спробі доступу до завантажувача (наприклад, за допомогою методу `initLoader()`), можна перевіити, чи існує завантажувач, вказаний за допомогою ідентифікатора. Якщо завантажувач 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 класс `CursorLoader`. Треба побудувати клас `CursorLoader` за допомогою його методу конструктора, для чого потрібен повний набір інформації, яка необхідна для виконання запиту до `ContentProvider`, зокрема:

– `uri` – URI контенту, який необхідно отримати;

– `projection` – список стовпців, які будуть повернуті; при передачі `null` будуть повернуті всі стовпці, а це неефективно;

– `selection` – фільтр, який оголошує, які рядки повертати, відформатований у вигляді пропозиції SQL WHERE (за винятком самого WHERE); при передачі `null` будуть повернуті всі рядки для даного URI;

– `selectionArgs` – у вибірку можна включити символи «?», які будуть замінені значеннями з selectionArgs в порядку проходження в вибірці. Значення будуть прив'язані як рядки;

– `sortOrder` – порядок розташування рядків, відформатований у вигляді пропозиції SQL ORDER BY (за винятком самого ORDER BY). При передачі `null` буде використовуватися стандартний порядок сортування, а тому, список, можливо, буде несортованим.

Наприклад:

```
// If non-null, this is the current filter the user has provided.
String mCurFilter;
...
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    // This is called when a new Loader needs to be created.  This
    // sample only has one Loader, so we don't care about the ID.
    // First, pick the base URI to use depending on whether we are
    // currently filtering.
    Uri baseUri;
    if (mCurFilter != null) {
        baseUri = Uri.withAppendedPath(Contacts.CONTENT_FILTER_URI,
                  Uri.encode(mCurFilter));
    } else {
        baseUri = Contacts.CONTENT_URI;
    }

    // Now create and return a CursorLoader that will take care of
    // creating a Cursor for the data being displayed.
    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");
}
```

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 вони скоро будуть замінені). 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 з `CursorLoader`, не слід викликати `close()` самостійно. Якщо курсор розміщується в `CursorAdapter`, слід використовувати метод `swapCursor()` з тим, щоб старий `Cursor` не закрився. Наприклад:

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

public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
    // Swap the new cursor in.  (The framework will take care of
closing the
    // old cursor once we return.)
    mAdapter.swapCursor(data);
}
```

Метод onLoaderReset. Цей метод викликається, коли стан створеного раніше завантажувача скидається, в результаті чого його дані губляться. Цей зворотний виклик дозволяє дізнатися, коли дані ось-ось будуть вивільнені, з тим щоб можна було видалити своє посилання на них.

Дана реалізація викликає `swapCursor()` зі значенням `null`:

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

public void onLoaderReset(Loader<Cursor> loader) {
    // This is called when the last Cursor provided to onLoadFinished()
    // above is about to be closed.  We need to make sure we are no
    // longer using it.
    mAdapter.swapCursor(null);
}
```

Приклад. Як приклад далі наведена повна реалізація фрагмента `Fragment`, який відображає `ListView` з результатами запиту до постачальника такого контенту, як контакти. Для управління запитом до постачальника використовується клас `CursorLoader`.

Щоб додаток міг звертатися до контактів користувача, як показано в цьому прикладі, в його маніфесті повинен бути присутнім дозвіл `READ_CONTACTS`.

```
public static class CursorLoaderListFragment extends ListFragment
        implements OnQueryTextListener, LoaderManager.LoaderCallbacks<Cursor> {

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

    // If non-null, this is the current filter the user has provided.
    String mCurFilter;

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

        // Give some text to display if there is no data.  In a real
        // application this would come from a resource.
        setEmptyText("No phone numbers");

        // We have a menu item to show in action bar.
        setHasOptionsMenu(true);
```

```
        // Create an empty adapter we will use to display the loaded data.
        mAdapter = new SimpleCursorAdapter(getActivity(),
                android.R.layout.simple_list_item_2, null,
                new String[] { Contacts.DISPLAY_NAME, Contacts.CONTACT_STATUS},
                new int[] { android.R.id.text1, android.R.id.text2 }, 0);
        setListAdapter(mAdapter);

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

    @Override public void onCreateOptionsMenu(Menu menu, MenuInflater inflater){
        // Place an action bar item for searching.
        MenuItem item = menu.add("Search");
        item.setIcon(android.R.drawable.ic_menu_search);
        item.setShowAsAction(MenuItem.SHOW_AS_ACTION_IF_ROOM);
        SearchView sv = new SearchView(getActivity());
        sv.setOnQueryTextListener(this);
        item.setActionView(sv);
    }
    public boolean onQueryTextChange(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;
    }

    @Override public boolean onQueryTextSubmit(String query) {
        // Don't care about this.
        return true;
    }

    @Override public void onListItemClick(ListView l, View v, int position, long id) {
        // Insert desired behavior here.
        Log.i("FragmentComplexList", "Item clicked: " + id);
    }

    // These are the Contacts rows that we will retrieve.
    static final String[] CONTACTS_SUMMARY_PROJECTION = new String[] {
        Contacts._ID,
        Contacts.DISPLAY_NAME,
        Contacts.CONTACT_STATUS,
        Contacts.CONTACT_PRESENCE,
        Contacts.PHOTO_ID,
        Contacts.LOOKUP_KEY,
    };
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
        // This is called when a new Loader needs to be created.  This
        // sample only has one Loader, so we don't care about the ID.
        // First, pick the base URI to use depending on whether we are
        // currently filtering.
        Uri baseUri;
        if (mCurFilter != null) {
            baseUri = Uri.withAppendedPath(Contacts.CONTENT_FILTER_URI,
                    Uri.encode(mCurFilter));
        } else {
            baseUri = Contacts.CONTENT_URI;
        }

        // Now create and return a CursorLoader that will take care of
        // creating a Cursor for the data being displayed.
        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");
```

```
    }

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

    public void onLoaderReset(Loader<Cursor> loader) {
        // This is called when the last Cursor provided to onLoadFinished()
        // above is about to be closed.  We need to make sure we are no
        // longer using it.
        mAdapter.swapCursor(null);
    }
}
```

Інші приклади. В ApiDemos є кілька різних прикладів, які ілюструють використання завантажувачів:

- `LoaderCursor` – повна версія показаного вище фрагмента;
- `LoaderThrottle` – приклад того, як використовувати регулювання для скорочення числа запитів, які виконуються постачальником контенту при зміні його даних.

4.4. Постачальник календаря

Постачальник календаря являє собою репозиторій для подій календаря користувача. API постачальника календаря дозволяє запитувати, вставляти, оновлювати і видаляти календарі, події, учасників, нагадування і т. д.

API постачальника календаря може використовуватися як додатки, так і адаптери синхронізації. Правила залежать від типу програми, яка виконує виклики. У цьому розділі головним чином розглядається використання API постачальника календаря як додатоку.

Зазвичай, щоб переглянути або записати дані календаря, в маніфесті додатка повинні бути включені відповідні дозволи. Щоб спростити виконання часто використовуваних операцій, в постачальника календаря передбачений набір намірів. Ці наміри дозволяють користувачам переходити в додаток календаря для вставки, перегляду і редагування подій. Користувач після взаємодії з календарем повертається у вихідний додаток. Тому вашому додатоку не треба запитувати дозволу, а також надавати призначений для користувача інтерфейс для перегляду або створення подій.

4.4.1. Основи

Постачальники контенту зберігають в собі дані і надають до них доступ для додатків. Постачальники контенту, запропоновані платформою Android (включаючи постачальник календаря), зазвичай подають дані у вигляді набору таблиць, в основі яких лежить модель реляційної бази даних. Кожен ря-

док в такій таблиці являє собою запис, а кожен стовпець – дані певного типу і значення. Завдяки API постачальника календаря програми та адаптери синхронізації отримують доступ на читання / запис до таблиць в базі даних, в яких наведені дані календаря користувача.

Кожен постачальник календаря надає загальнодоступний URI (упакований в об'єкті `Uri`), який служить унікальним ідентифікатором свого набору даних. Постачальник контенту, який керує кількома наборами даних (кілька таблиць), надає окремий URI для кожного набору. Всі URI постачальників починаються з рядка: //. Він визначає дані, які знаходяться під управлінням постачальника контенту. Постачальник календаря задає константи для URI кожного зі своїх класів (таблиці). Такі URI мають формат `<class>.CONTENT_URI.` Наприклад, `Events.CONTENT_URI.`

На рис. 4.1 зображено графічне подання моделі даних постачальника календаря. На ньому наведено основні таблиці і поля, які пов'язують їх один з одним.

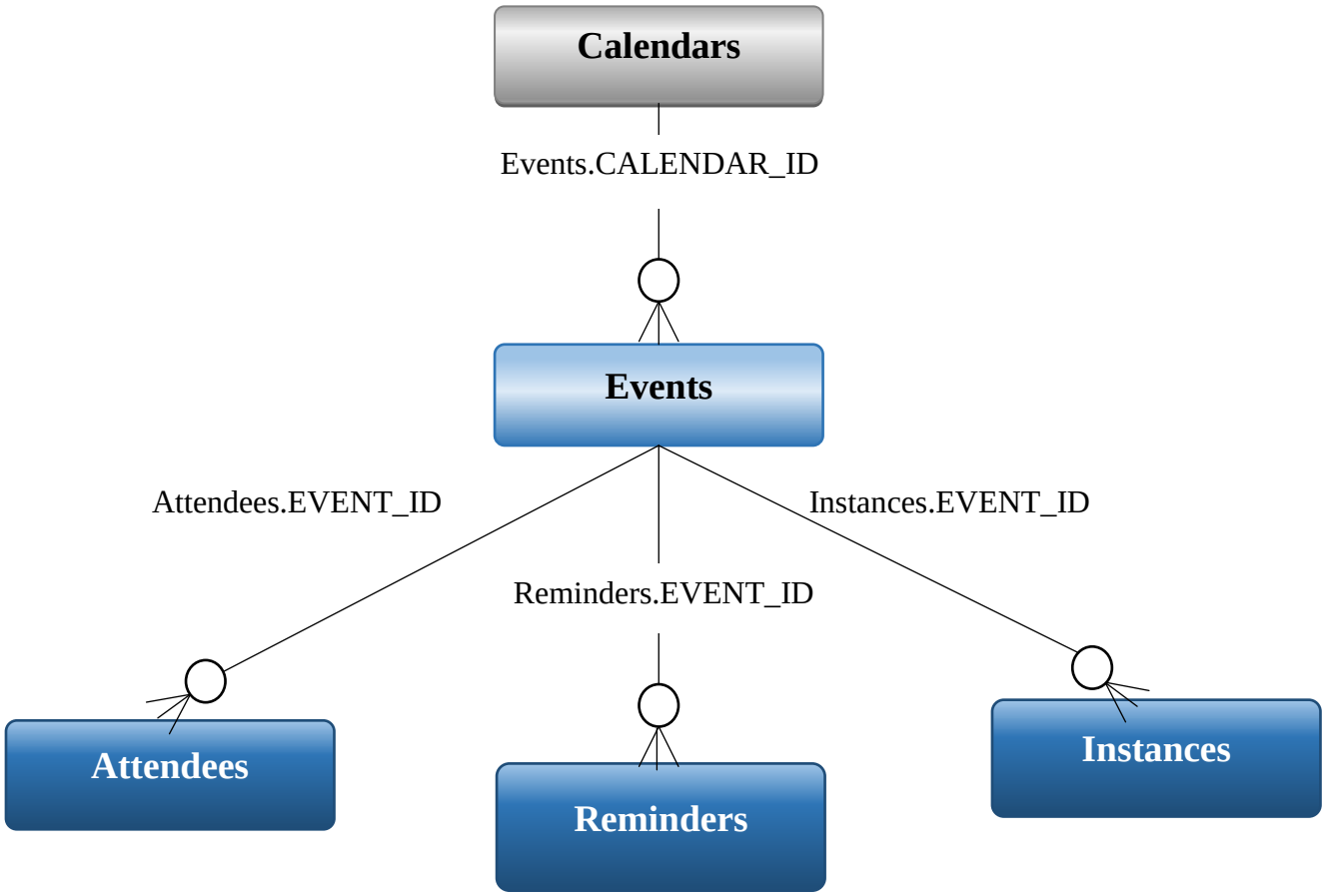

Рисунок 4.1 – Модель даних постачальника календаря

У користувача може бути кілька календарів, причому вони можуть бути пов'язані з акаунтами різних типів (Google Календар, Обмін і т. д.).

Клас `CalendarContract` визначає модель даних календаря і інформацію, що відноситься до подій. Ці дані зберігаються в різних таблицях, зазначених нижче.

Таблиця 4.4 – Модель даних календаря і події

Таблиця (клас) CalendarContract	**Опис**
1	2
`.Calendars`	У цій таблиці знаходиться інформація про календарі. У кожному рядку цієї таблиці наведені відомості про окремий календар, наприклад, його назва, колір, інформація про синхронізацію і т. д.
`.Events`	У цій таблиці знаходиться інформація про події. У кожному рядку цієї таблиці міститься інформація про окрему подію, наприклад, заголовок події, місце проведення, час початку, час завершення і т. д. Подія може бути одноразовою або повторюваною. Відомості про учасників, нагадування і розширені властивості зберігаються в окремих таблицях. У кожній з них є цілочисельна змінна `EVENT_ID`, яка посилається на об'єкт `_ID` в таблиці подій
`.Instances`	У цій таблиці містяться дані про час початку і закінчення кожної повторюваної події. У кожному рядку цієї таблиці наведено одне повторення події. Одноразові події зіставляються з повтореннями один до одного. Для повторюваних подій автоматично створюються кілька рядків, що відповідають декільком повторенням події.
`.Attendees`	У цій таблиці знаходиться інформація про учасників (гостей). У кожному рядку цієї таблиці вказаний один гість. У ній вказується тип гостя і інформація про те, чи відвідає він подію
`.Reminders`	У цій таблиці знаходиться дані повідомлень або оповіщень. У кожному рядку цієї таблиці вказано одне повідомлення або оповіщення. Для однієї події можна створити кілька нагадувань. Максимальна кількість таких нагадувань для події задається за допомогою цілочисельної змінної `MAX_REMINDERS`, значення якої задає адаптер синхронізації, що володіє вказаним календарем. Нагадування задається в хвилинах до початку події і має метод, який визначає порядок повідомлення користувача

API постачальника календаря забезпечує достатню гнучкість і ефективність. У той же час важливо надати інтерфейс, який буде зручним для користувача, і забезпечити захист цілісності календаря і його даних. Тому існує ряд моментів, які слід враховувати при використанні цього API.

1. Вставка, оновлення і перегляд подій календаря. Щоб вставити, змінити і переглянути події безпосередньо з постачальника календаря, потрібні відповідні дозволи. Однак якщо не планується створювати повнофункціональний додаток календаря або адаптер синхронізації, запитувати такі дозволи не обов'язково. Замість цього можна використовувати намір, підтримуваний додатком «Календар» Android, для обробки операцій читання і запису в цьому додатка. При використанні намірів ваш додаток відправляє користувачам додаток «Календар» для виконання необхідної операції в попередньо заповненій формі. По її завершенні вони повертаються до додатку. Завдяки можливості виконання часто використовуваних операцій через додаток «Календар», для користувачів забезпечується однаковий і функціональний користувацький інтерфейс. Рекомендуємо використовувати саме такий підхід.

2. Адаптери синхронізації. Адаптер синхронізації синхронізує дані календаря на пристрої користувача з даними на сервері або в іншому джерелі даних. У таблицях `CalendarContract.Calendars` і `CalendarContract.Events` є стовпці, зарезервовані для адаптерів синхронізації. Ні постачальник, ні програми не повинні змінювати їх. Фактично вони приховані доти, поки адаптер синхронізації не почне використовувати їх.

4.4.2. Дозволи користувачів

Щоб прочитати дані календаря, у файл маніфесту додатка необхідно включати дозвіл `READ_CALENDAR`. Також в нього слід включити дозвіл `WRITE_CALENDAR` для видалення, вставки або поновлення даних календаря:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"...>
    <uses-sdk android:minSdkVersion="14" />
    <uses-permission an-
droid:name="android.permission.READ_CALENDAR" />
    <uses-permission an-
droid:name="android.permission.WRITE_CALENDAR" />
    ...
</manifest>
```

4.4.3. Таблиця календарів

У таблиці `CalendarContract.Calendars` містяться докладні відомості про кожний окремий календар (табл. 4.5). Виконувати запис у зазначених нижче стовпцях цієї таблиці може і додаток, і адаптер синхронізації. Повний список підтримуваних полів наведено в довідці по класу `CalendarContract.Calendars`.

Таблиця 4.5 – Відомості про календарі

Константа	Опис
NAME	Назва календаря
CALENDAR_DISPLAY_NAME	Назва цього календаря, яка відображається для користувача
VISIBLE	Логічне значення, що показує, чи вибраний календар для відображення. Значення «0» вказує на те, що події, пов'язані з цим календарем, не відображаються. Значення «1» вказує на те, що події, пов'язані з цим календарем, відображаються. Це значення впливає на створення рядків в таблиці `CalendarContract.Instances`
SYNC_EVENTS	Логічне значення, що позначає, чи слід синхронізувати календар і зберігати наявні в ньому події на пристрої. Значення «0» вказує, що не слід синхронізувати цей календар або зберігати наявні в ньому події на пристрої. Значення «1» вказує, що цей календар слід синхронізувати і зберігати наявні в ньому події на пристрої

Запит календаря. Нижче наведено приклад того, як отримати календарі, якими володіє певний користувач. Для простоти демонстрації операція запиту в цьому прикладі знаходиться в потоці призначеного для користувача інтерфейсу («основний потік»). На практиці це слід робити в асинхронному потоці, а не в основному. Якщо ж ви не тільки зчитуєте дані, але і вносите в них зміни, зверніться до довідки по класу `AsyncQueryHandler.`

```
// Масив проекції. Створення індексів для цього масиву, замість
// динамічних пошуків підвищує продуктивність.
public static final String[] EVENT_PROJECTION = new String[] {
    Calendars._ID,                           // 0
    Calendars.ACCOUNT_NAME,                  // 1
    Calendars.CALENDAR_DISPLAY_NAME,         // 2
    Calendars.OWNER_ACCOUNT                  // 3
};

// Індекси масиву проекції вище.
private static final int PROJECTION_ID_INDEX = 0;
private static final int PROJECTION_ACCOUNT_NAME_INDEX = 1;
```

```
private static final int PROJECTION_DISPLAY_NAME_INDEX = 2;
private static final int PROJECTION_OWNER_ACCOUNT_INDEX = 3;
```

В наступній частині прикладу створюється запит. За допомогою вибору визначаються критерії для запиту. У цьому прикладі виконується пошук календарів з такими значеннями параметрів: `ACCOUNT_NAME` – sampleuser@google.com, `ACCOUNT_TYPE` – com.google та `OWNER_ACCOUNT` – sampleuser@google.com. Для перегляду всіх переглянутих користувачем календарів, а не тільки наявних у нього, не вказуйте параметр `OWNER_ACCOUNT`. Запит повертає об'єкт `Cursor`, який можна використовувати для перебору результатів.

Повернутий запит до бази даних.

```
// Виконати запит
Cursor cur = null;
ContentResolver cr = getContentResolver();
Uri uri = Calendars.CONTENT_URI;
String selection = "((" + Calendars.ACCOUNT_NAME + " = ?) AND ("
                        + Calendars.ACCOUNT_TYPE + " = ?) AND ("
                        + Calendars.OWNER_ACCOUNT + " = ?))";
String[] selectionArgs = new String[] {"sampleuser@gmail.com",
"com.google",
        "sampleuser@gmail.com"};
// Відправити запит і отримати об'єкт курсору назад.
cur = cr.query(uri, EVENT_PROJECTION, selection, selectionArgs,
null);
```

У наступному розділі коду виконується покроковий огляд набору результатів за допомогою курсора. У ньому використовуються константи, які були задані на початку прикладу, для отримання значень для кожного з полів.

```
// Використовуйте курсор для покрокового повернутих записів
while (cur.moveToNext()) {
    long calID = 0;
    String displayName = null;
    String accountName = null;
    String ownerName = null;

    // Отримати значення полів
    calID = cur.getLong(PROJECTION_ID_INDEX);
    displayName = cur.getString(PROJECTION_DISPLAY_NAME_INDEX);
    accountName = cur.getString(PROJECTION_ACCOUNT_NAME_INDEX);
    ownerName = cur.getString(PROJECTION_OWNER_ACCOUNT_INDEX);

    // Зробіти що-небудь зі значеннями...

   ...
}
```

4.4.4. Навіщо необхідно вказувати параметр ACCOUNT_TYPE?

При створенні запиту `Calendars.ACCOUNT_NAME` необхідно також вказати `Calendars.ACCOUNT_TYPE`. Це необхідно зробити з огляду на те, що вказаний акаунт вважається унікальним тільки тоді, коли для нього вказані і параметр `ACCOUNT_NAME`, й параметр `ACCOUNT_TYPE`. Параметр `ACCOUNT_TYPE` в рядку позначає структуру перевірки існування акаунта, яка використовувалася при реєстрації облікового запису за допомогою `AccountManager`. Існує також особливий тип акаунтів, званий `ACCOUNT_TYPE_LOCAL`. Він використовується для календарів, які не пов'язані з членством пристрою. Акаунти `ACCOUNT_TYPE_LOCAL` не синхронізуються.

Зміна календаря. Щоб оновити календар, можна вказати `_ID` календаря: або у вигляді ідентифікатора, доданого до URI (`withAppendedId()`), або як перший елемент виділення. Виділення повинне починатися з «`_id=?`», а перший аргумент `selectionArg` повинен бути `_ID` календаря. Також для виконання оновлень можна закодувати ідентифікатор в URI. У цьому прикладі для редагування Вашого календаря використовується підхід (`withAppendedId()`):

```
private static final String DEBUG_TAG = "MyActivity";
...
long calID = 2;
ContentValues values = new ContentValues();
// Нова дисплей назва для календаря
values.put(Calendars.CALENDAR_DISPLAY_NAME, "Trevor's Calendar");
Uri updateUri = ContentUris.withAppendedId(Calendars.CONTENT_URI,
calID);
int rows = getContentResolver().update(updateUri, values, null,
null);
Log.i(DEBUG_TAG, "Rows updated: " + rows);
```

Вставка календаря. Для управління календарями в основному використовуються адаптери синхронізації, тому нові календарі слід вставляти виключно як адаптер синхронізації. Здебільшого додатки можуть вносити в календарі тільки поверхневі зміни, такі, як зміна імен. Якщо додатку потрібно створити локальний календар, це можна зробити шляхом вставки календаря у вигляді адаптера синхронізації за допомогою параметра `ACCOUNT_TYPE` типу `ACCOUNT_TYPE_LOCAL.ACCOUNT_TYPE_LOCAL` являє собою особливий тип акаунтів для календарів. Календарі цього типу не синхронізуються з сервером.

4.4.5. Таблиця подій

У таблиці `CalendarContract.Events` містяться докладні відомості про кожну окремому подію. Щоб отримати можливість додавати, оновлювати або видаляти події, у файл маніфесту додатка необхідно включати дозвіл `WRITE_CALENDAR`.

Виконувати запис в зазначені нижче стовпці цієї таблиці можуть і додаток, і адаптер синхронізації. Повний список підтримуваних полів наведено в довідці по класу `CalendarContract.Events`.

Таблиця 4.6 – Відомості про календар

Константа	Опис
`CALENDAR_ID`	`_ID` календаря, до якого належить подія
`ORGANIZER`	Адреса ел. пошти організатора (власника) події
`TITLE`	Назва події
`EVENT_LOCATION`	Місце проведення
`DESCRIPTION`	Опис події
`DTSTART`	Час початку події за UTC (в мілісекундах) від точки відліку
`DTEND`	Час закінчення події по UTC (в мілісекундах) від точки відліку
`EVENT_TIMEZONE`	Часовий пояс події
`EVENT_END_TIMEZONE`	Часовий пояс для часу закінчення події
`DURATION`	Тривалість події в форматі RFC5545. Наприклад, значення "PT1H" означає, що подія має тривати одну годину, а значення "P2W" вказує на тривалість в 2 тижні
`ALL_DAY`	Значення «1» означає, що ця подія триває весь день за місцевим часовим поясом. Значення «0» вказує на те, що це регулярна подія, яке може початися і завершитися в будь-який час протягом дня
`RRULE`	Правило повторення для формату події. Наприклад, `"FREQ=WEEKLY;COUNT=10;WKST=SU"`. З іншими прикладами можна ознайомитися тут
`RDATE`	Дати повторення події. Зазвичай `RDATE` використовується разом з `RRULE` для визначення агрегованого набору повторюваних подій. Додаткові відомості наведено в спецификации RFC5545
`AVAILABILITY`	Подія вважається як зайнята або як вільний час, доступний для планування
`GUESTS_CAN_MODIFY`	Вказує, чи можуть гості вносити зміни в подію
`GUESTS_CAN_INVITE_OTHERS`	Вказує, чи можуть гості запрошувати інших гостей
`GUESTS_CAN_SEE_GUESTS`	Вказує, чи можуть гості переглядати список учасників

Додавання подій. Коли ваш додаток вставляє нову подію, ми рекомендуємо використовувати намір `INSERT`. Однак за необхідності ви можете вставляти події безпосередньо.

Правила, якими слід керуватися для вставки нової події, такі:

1. Необхідно вказати `CALENDAR_ID` й `DTSTART`.

2. Необхідно вказати `EVENT_TIMEZONE`. Щоб отримати список встановлених в системі ідентифікаторів часових поясів, скористайтеся методом `getAvailableIDs()`. Зверніть увагу, що це правило не застосовується при вставці події за допомогою наміру `INSERT`, – в цьому випадку використовується часовий пояс за замовчуванням.

3. Для одноразових подій необхідно вказати `DTEND`.

4. Для повторюваних подій необхідно вказати `DURATION` на додаток до `RRULE` чи `RDATE`. Зверніть увагу, що це правило не застосовується при вставці події за допомогою наміру `INSERT`, – в цьому випадку можна використовувати `RRULE` в поєднанні з `DTSTART` й `DTEND`; крім того, додаток «Календар» автоматично перетворює вказаний період тривалості.

Нижче наведено приклад вставки події. Для простоти все це виконується в потоці призначеного для користувача інтерфейсу. На практиці ж всі вставки і оновлення слід виконувати в асинхронному потоці, щоб перемістити операцію в фоновому потоці. Додаткові відомості наведено в довідці по `AsyncQueryHandler`.

```
long calID = 3;
long startMillis = 0;
long endMillis = 0;
Calendar beginTime = Calendar.getInstance();
beginTime.set(2012, 9, 14, 7, 30);
startMillis = beginTime.getTimeInMillis();
Calendar endTime = Calendar.getInstance();
endTime.set(2012, 9, 14, 8, 45);
endMillis = endTime.getTimeInMillis();
...

ContentResolver cr = getContentResolver();
ContentValues values = new ContentValues();
values.put(Events.DTSTART, startMillis);
values.put(Events.DTEND, endMillis);
values.put(Events.TITLE, "Jazzercise");
values.put(Events.DESCRIPTION, "Group workout");
values.put(Events.CALENDAR_ID, calID);
values.put(Events.EVENT_TIMEZONE, "America/Los_Angeles");
Uri uri = cr.insert(Events.CONTENT_URI, values);
```

```
// get the event ID that is the last element in the Uri
long eventID = Long.parseLong(uri.getLastPathSegment());
//
// ... do something with event ID
//
//
```

Примітка. Нижче демонструється, як в прикладі коду виконується захоплення ідентифікатора події після створення цієї події. Це найпростіший спосіб отримати ідентифікатор події. Найчастіше ідентифікатор події необхідний для виконання інших дій з календарем – наприклад, для додавання учасників або нагадувань про подію.

Оновлення подій. Коли ваш додаток хоче надати користувачеві можливість змінити подію, ми рекомендуємо використовувати намір `EDIT`. Однак за необхідності можна редагувати події безпосередньо. Щоб оновити подію, можна вказати _ID події: або у вигляді ідентифікатора, доданого до URI (`withAppendedId()`), або як перший елемент виділення. Виділення повинне починатися з `"_id=?"`, а першим аргументом `selectionArg` повинен бути `_ID` події. Також можна оновлювати виділення без ідентифікаторів. Нижче наведений приклад поновлення події. Це приклад зміни назви події за допомогою методу `withAppendedId()`:

```
private static final String DEBUG_TAG = "MyActivity";
...
long eventID = 188;
...
ContentResolver cr = getContentResolver();
ContentValues values = new ContentValues();
Uri updateUri = null;
// Нова назва для події
values.put(Events.TITLE, "Kickboxing");
updateUri = ContentUris.withAppendedId(Events.CONTENT_URI,
eventID);
int rows = getContentResolver().update(updateUri, values, null,
null);
Log.i(DEBUG_TAG, "Rows updated: " + rows);
```

Видалення подій. Видалити запис можна по його `_ID`, який доданий як ідентифікатор до URI, або за допомогою стандартного виділення. У разі використання доданого ідентифікатора неможливо виконати і виділення. Існує дві версії операції видалення: для додатка і для адаптера синхронізації. При видаленні для додатка в стовпці *deleted* встановлюється значення «1». Цей пра-

пор повідомляє адаптера синхронізації про те, що рядок був видалений і інформацію про видалення слід передати серверу. При видаленні для адаптера синхронізації подія видаляється з бази даних разом з усіма пов'язаними з нею даними. Нижче наведено приклад видалення події для додатка по його _ID:

```
private static final String DEBUG_TAG = "MyActivity";
...
long eventID = 201;
...
ContentResolver cr = getContentResolver();
ContentValues values = new ContentValues();
Uri deleteUri = null;
deleteUri = ContentUris.withAppendedId(Events.CONTENT_URI, eventID);
int rows = getContentResolver().delete(deleteUri, null, null);
Log.i(DEBUG_TAG, "Rows deleted: " + rows);
```

4.4.6. Таблиця учасників

У кожному рядку таблиці CalendarContract.Attendees вказано один учасник або гість події. При виклику методу query() повертається список учасників для події з заданим EVENT_ID. Цей EVENT_ID повинен відповідати _ID певної події.

У табл. 4.7 вказані поля, доступні для запису. При вставці нового учасника необхідно вказати всі ці поля, крім ATTENDEE_NAME.

Таблиця 4.7 – Таблиця участників

Константа	Опис
EVENT_ID	Ідентифікатор події
ATTENDEE_NAME	Ім'я учасника
ATTENDEE_EMAIL	Адреса ел. пошти учасника
ATTENDEE_RELATIONSHIP	Зв'язок учасника з подією. Один з наступних: • RELATIONSHIP_ATTENDEE • RELATIONSHIP_NONE • RELATIONSHIP_ORGANIZER • RELATIONSHIP_PERFORMER • RELATIONSHIP_SPEAKER
ATTENDEE_TYPE	Тип учасника. Один з наступних: • TYPE_REQUIRED • TYPE_OPTIONAL
ATTENDEE_STATUS	Статус відвідування події учасником. Один з наступних: ATTENDEE_STATUS_ACCEPTED ATTENDEE_STATUS_DECLINED ATTENDEE_STATUS_INVITED ATTENDEE_STATUS_NONE ATTENDEE_STATUS_TENTATIVE

Додавання учасників. Нижче наведено приклад додавання одного учасника події. Зверніть увагу, що потрібно в обов'язковому порядку вказати EVENT_ID.

```
long eventID = 202;
...
ContentResolver cr = getContentResolver();
ContentValues values = new ContentValues();
values.put(Attendees.ATTENDEE_NAME, "Trevor");
values.put(Attendees.ATTENDEE_EMAIL, "trevor@example.com");
values.put(Attendees.ATTENDEE_RELATIONSHIP, At-
tendees.RELATIONSHIP_ATTENDEE);
values.put(Attendees.ATTENDEE_TYPE, Attendees.TYPE_OPTIONAL);
values.put(Attendees.ATTENDEE_STATUS, At-
tendees.ATTENDEE_STATUS_INVITED);
values.put(Attendees.EVENT_ID, eventID);
Uri uri = cr.insert(Attendees.CONTENT_URI, values);
```

Додавання подій. У кожному рядку таблиці CalendarContract.Reminders вказано одне нагадування про подію. При виклику методу query() повертається список нагадування для події з заданим EVENT_ID.

У табл. 4.8 вказані поля, доступні для запису. При вставці нового нагадування необхідно вказати всі ці поля. Зверніть увагу, що адаптери синхронізації задають типи нагадувань, які вони підтримують, в таблиці CalendarContract.Calendars. Докладну інформацію див. в довідці по ALLOWED_REMINDERS.

Таблиця 4.8 – Таблиця нагадувань

Константа	Опис
EVENT_ID	Ідентифікатор події
MINUTES	Час спрацювання повідомлення (в хвилинах) до початку події
METHOD	Метод повідомлення, заданий на сервері. Один з наступного: • METHOD_ALERT • METHOD_DEFAULT • METHOD_EMAIL • METHOD_SMS

Додавання нагадувань. Нижче наведено приклад додавання нагадування в подію. Нагадування спрацьовує за 15 хвилин до початку події:

```
long eventID = 221;
...
```

```
ContentResolver cr = getContentResolver();
ContentValues values = new ContentValues();
values.put(Reminders.MINUTES, 15);
values.put(Reminders.EVENT_ID, eventID);
values.put(Reminders.METHOD, Reminders.METHOD_ALERT);
Uri uri = cr.insert(Reminders.CONTENT_URI, values);
```

4.4.7. Таблиця екземплярів

У таблиці `CalendarContract.Instances` (табл. 4.9) містяться дані про час початку і закінчення повторень події. У кожному рядку цієї таблиці наведено одне повторення події. Таблиця екземплярів недоступна для запису; вона надає тільки можливість запитувати повторення подій.У ній перераховані деякі з полів, які можна запросити для екземпляра. Зверніть увагу, що часовий пояс задається параметрами `KEY_TIMEZONE_TYPE` та `KEY_TIMEZONE_INSTANCES`.

Таблиця 4.9 – Таблиця екземплярів

Константа	Опис
`BEGIN`	Час початку екземпляра в форматі UTC (в мілісекундах)
`END`	Час закінчення екземпляра в форматі UTC (в мілісекундах)
`END_DAY`	День закінчення екземпляра за юліанським календарем щодо часового поясу додатка «Календар»
`END_MINUTE`	Хвилина закінчення екземпляра, обчислена від півночі за часовим поясом додатка «Календар»._ID події для цього екземпляра
`EVENT_ID`	День початку екземпляра за юліанським календарем щодо часового поясу додатка «Календар»
`START_DAY`	Хвилина початку екземпляра, обчислена від півночі за часовим поясом додатка «Календар»
`START_MINUTE`	День закінчення екземпляра за юліанським календарем щодо часового поясу додатка «Календар»

Запит таблиці екземплярів. Щоб запросити таблицю екземплярів, необхідно вказати проміжок часу для запиту в URI. У цьому прикладі `CalendarContract.Instances` отримує доступ до поля `TITLE` за допомогою своєї реалізації інтерфейсу `CalendarContract.EventsColumns`. Іншими словами, `TITLE` повертається за допомогою звернення до бази даних, а не шляхом запиту таблиці `CalendarContract.Instances` з необробленими даними:

```
private static final String DEBUG_TAG = "MyActivity";
public static final String[] INSTANCE_PROJECTION = new String[] {
```

```
    Instances.EVENT_ID,      // 0
    Instances.BEGIN,         // 1
    Instances.TITLE          // 2
  };

// Індекси масиву проекцій вище.
private static final int PROJECTION_ID_INDEX = 0;
private static final int PROJECTION_BEGIN_INDEX = 1;
private static final int PROJECTION_TITLE_INDEX = 2;
...

// Вкажіть діапазон дат, який ви хочете шукати для повторювання
// екземплярів подій
Calendar beginTime = Calendar.getInstance();
beginTime.set(2011, 9, 23, 8, 0);
long startMillis = beginTime.getTimeInMillis();
Calendar endTime = Calendar.getInstance();
endTime.set(2011, 10, 24, 8, 0);
long endMillis = endTime.getTimeInMillis();

Cursor cur = null;
ContentResolver cr = getContentResolver();

// Ідентифікатор повторюваної події, екземпляри якої ви шукаєте
// в таблиці Примірники
String selection = Instances.EVENT_ID + " = ?";
String[] selectionArgs = new String[] {"207"};

// Побудувати запит з бажаним діапазоном дат.
Uri.Builder builder = Instances.CONTENT_URI.buildUpon();
ContentUris.appendId(builder, startMillis);
ContentUris.appendId(builder, endMillis);

// Відправити запит
cur =  cr.query(builder.build(),
    INSTANCE_PROJECTION,
    selection,
    selectionArgs,
    null);

while (cur.moveToNext()) {
    String title = null;
    long eventID = 0;
    long beginVal = 0;

    // Отримати значення полів
    eventID = cur.getLong(PROJECTION_ID_INDEX);
    beginVal = cur.getLong(PROJECTION_BEGIN_INDEX);
    title = cur.getString(PROJECTION_TITLE_INDEX);

    // Зробити що-небудь зі значеннями.
```

```
    Log.i(DEBUG_TAG, "Event:  " + title);
    Calendar calendar = Calendar.getInstance();
    calendar.setTimeInMillis(beginVal);
    DateFormat formatter = new SimpleDateFormat("MM/dd/yyyy");
    Log.i(DEBUG_TAG, "Date: " + format-
ter.format(calendar.getTime()));
    }
 }
```

4.4.8. Наміри календаря

Вашому додатку не потрібно запитувати дозволи на читання і запис даних календаря. Замість цього можна використовувати наміри, підтримувані додатком «Календар» Android, для обробки операцій читання і запису в цьому додатку. У табл. 4.10 вказані наміри, підтримувані постачальником календаря, у табл. 4.11 вказані додаткові дані намірів, які підтримуються постачальником календаря.

Таблиця 4.11 – Додаткові дані наміру

Додаткові дані наміру	Опис
`Events.TITLE`	Назва події
`CalendarContract.EXTRA_EVENT_BEGIN_TIME`	Час початку події (в мілісекундах) від епохи
`CalendarContract.EXTRA_EVENT_END_TIME`	Час закінчення події (в мілісекундах) від епохи
`CalendarContract.EXTRA_EVENT_ALL_DAY`	Логічне значення, яке позначає, що ця подія на весь день. Значення може бути `true` чи `false`
`Events.EVENT_LOCATION`	Місце проведення події
`Events.DESCRIPTION`	Опис події
`Intent.EXTRA_EMAIL`	Адреси ел. пошти запрошених (через кому)
`Events.RRULE`	Правило повторення для події
`Events.ACCESS_LEVEL`	Вказує на те, чи є подія загальнодоступною або закритою
`Events.AVAILABILITY`	Подія вважається як зайнята або як вільний час, доступний для планування

У наступних розділах зазначено порядок використання даних намірів.

Таблиця 4.10 – Наміри календаря

Дія	URI	Опис	Додаткові дані
VIEW	content://com.android.calendar/time/<ms_since_epoch> Зіслатися на URI також можна за допомогою CalendarContract.CONTENT_URI	Відкриття календаря під час, заданий параметром <ms_since_epoch>.	Відсутні
VIEW	content://com.android.calendar/events/<event_id> Зіслатися на URI також можна за допомогою Events.CONTENT_URI	Перегляд події, зазначеної за допомогою <event_id>	CalendarContract.EXTRA_EVENT_BEGIN_TIME CalendarContract.EXTRA_EVENT_END_TIME
EDIT	content://com.android.calendar/events/<event_id> Зіслатися на URI також можна за допомогою Events.CONTENT_URI	Редагування події, зазначеної за допомогою <event_id>	CalendarContract.EXTRA_EVENT_BEGIN_TIME CalendarContract.EXTRA_EVENT_END_TIME
EDIT INSERT	content://com.android.calendar/events Зіслатися на URI також можна за допомогою Events.CONTENT_URI	Створення події	Будь-які з додаткових даних, вказаних в табл. 4.11

Використання наміру для вставки події. За допомогою наміру `INSERT` ваш додаток може відправляти завдання вставки події прямо в додаток «Календар». Завдяки цьому у файл маніфесту вашого додатка не потрібно включати дозвіл `WRITE_CALENDAR`.

Коли користувачі працюють з додатком, в якому використовується такий підхід, додаток відправляє їх в «Календар» для завершення додавання події. Намір `INSERT` використовує додаткові поля для попередньої вказівки у формі відомостей про подію в додатку «Календар». Після цього користувачі можуть скасувати подію, відредагувати форму або зберегти подію в своєму календарі.

Нижче подано фрагмент коду, в якому на 19 січня 2012 р. планується подія, яка буде проходити з 7:30 до 8:30:

```
Calendar beginTime = Calendar.getInstance();
beginTime.set(2012, 0, 19, 7, 30);
Calendar endTime = Calendar.getInstance();
endTime.set(2012, 0, 19, 8, 30);
Intent intent = new Intent(Intent.ACTION_INSERT)
        .setData(Events.CONTENT_URI)
        .putExtra(CalendarContract.EXTRA_EVENT_BEGIN_TIME,
beginTime.getTimeInMillis())
        .putExtra(CalendarContract.EXTRA_EVENT_END_TIME,
endTime.getTimeInMillis())
        .putExtra(Events.TITLE, "Yoga")
        .putExtra(Events.DESCRIPTION, "Group class")
        .putExtra(Events.EVENT_LOCATION, "The gym")
        .putExtra(Events.AVAILABILITY, Events.AVAILABILITY_BUSY)
        .putExtra(Intent.EXTRA_EMAIL, "ro-
wan@example.com,trevor@example.com");
startActivity(intent);
```

Однак слід врахувати деякі моменти, що стосуються цього прикладу коду:

- `Events.CONTENT_URI` задається в ньому як URI;
- для попередньої вказівки у формі відомостей про час події у ній використовуються додаткові поля `CalendarContract.EXTRA_EVENT_BEGIN_TIME` і `CalendarContract.EXTRA_EVENT_END_TIME`. Значення часу повинні бути вказані в форматі UTC та в мілісекундах від епохи;
- для надання списку учасників, розділених комами (їх адреси ел. пошти) ,у ньому використовується додаткове поле `Intent.EXTRA_EMAIL`.

Використання наміру для редагування події. Подію можна відредагувати безпосередньо. Завдяки наміру `EDIT` додаток, у якого немає дозволу, може делегувати редагування події додатку «Календар». По завершенні редагування події в додатку «Календар» користувачі повертаються у вихідний додаток.

Нижче подано приклад наміру, що задає новий заголовок для вказаної події і що дозволяє користувачам редагувати подію в додатку «Календар».

```
long eventID = 208;
Uri uri = ContentUris.withAppendedId(Events.CONTENT_URI,
eventID);
Intent intent = new Intent(Intent.ACTION_EDIT)
    .setData(uri)
    .putExtra(Events.TITLE, "My New Title");
startActivity(intent);
```

Використання наміру для перегляду даних календаря. Постачальник календаря дозволяє використовувати намір `VIEW` двома різними способами:

- через відкриття додатка «Календар» на певній даті;
- перегляд події.

Нижче подано приклад відкриття додатку «Календар» на певній даті:

```
// Дата-час, вказаний в мілісекундах з початку епохи.
long startMillis;
...
Uri.Builder builder = CalendarContract.CONTENT_URI.buildUpon();
builder.appendPath("time");
ContentUris.appendId(builder, startMillis);
Intent intent = new Intent(Intent.ACTION_VIEW)
    .setData(builder.build());
startActivity(intent);
```

Приклад відкриття події для його перегляду:

```
long eventID = 208;
...
Uri uri = ContentUris.withAppendedId(Events.CONTENT_URI,
eventID);
Intent intent = new Intent(Intent.ACTION_VIEW)
   .setData(uri);
startActivity(intent);
```

4.4.10. Адаптери синхронізації

Існують лише незначні відмінності в тому, як додаток і адаптер синхронізації отримують доступ до постачальника календаря:

- адаптеру синхронізації необхідно вказати, що він є таким, задавши для параметра CALLER_IS_SYNCADAPTER значення `true`;
- адаптеру синхронізації необхідно надати ACCOUNT_NAME і ACCOUNT_TYPE як параметри запиту в URI;

- адаптер синхронізації має доступ на запис до більшої кількості стовпців, ніж додаток або віджет. Наприклад, додаток може змінювати тільки деякі характеристики календаря, такі, як назва, ім'я, настройка видимості і синхронізація. Тоді як адаптер синхронізації має доступ не тільки до цих стовпців, а й до багатьох інших його характеристик, таких, як колір календаря, часовий пояс, рівень доступу, місце розташування і т. д. Однак адаптер синхронізації обмежений заданими їм параметрами `ACCOUNT_NAME` і `ACCOUNT_TYPE`.

Нижче подано метод, який можна використовувати, щоб отримати URI для використання разом з адаптером синхронізації:

```
 static Uri asSyncAdapter(Uri uri, String account, String
accountType) {
    return uri.buildUpon()

.appendQueryParameter(android.provider.CalendarContract.CALLER_IS
_SYNCADAPTER,"true")
        .appendQueryParameter(Calendars.ACCOUNT_NAME, account)
        .appendQueryParameter(Calendars.ACCOUNT_TYPE,
accountType).build();
 }
```

4.5. Постачальник контактів

Постачальник контактів являє собою ефективний і гнучкий компонент Android, який керує центральним репозиторієм пристрою, в якому зберігаються призначені для користувача дані. Постачальник контактів – це джерело даних, які відображаються в меню «Контакти» на вашому пристрої. Також можна отримати доступ до цих даних в своєму власному додатку і організувати обмін такими даними між пристроєм і службами в Інтернеті. Постачальник взаємодіє з широким набором джерел даних і намагається організувати управління якомога більшим набіром даних про кожну людину, тому організація постачальника досить складна. З цієї причини API постачальника містить широкий набір класів-контрактів та інтерфейси, що відповідають як за отримання даних, так і за їх зміну.

У цьому керівництві розглядаються такі питання:

- основна структура постачальника;
- порядок отримання даних від постачальника;
- порядок зміни даних в постачальнику;
- порядок запису адаптера синхронізації для синхронізації даних, отриманих з вашого сервера, з даними в постачальнику контактів.

При вивченні даного матеріалу мається на увазі, що ви вже знайомі з основами постачальників контенту Android. Приклад адаптера синхронізації є прикладом використання такого додатка для обміну даними між постачальником контактів і додатком, розміщеним в веб-службах Google.

4.5.1. Структура постачальника контактів

Постачальник контактів являє собою постачальник контенту Android. Він містить в собі три типи даних про користувача, кожен з яких вказаний в окремій таблиці, що надається постачальником, як показано на рис. 4.2.

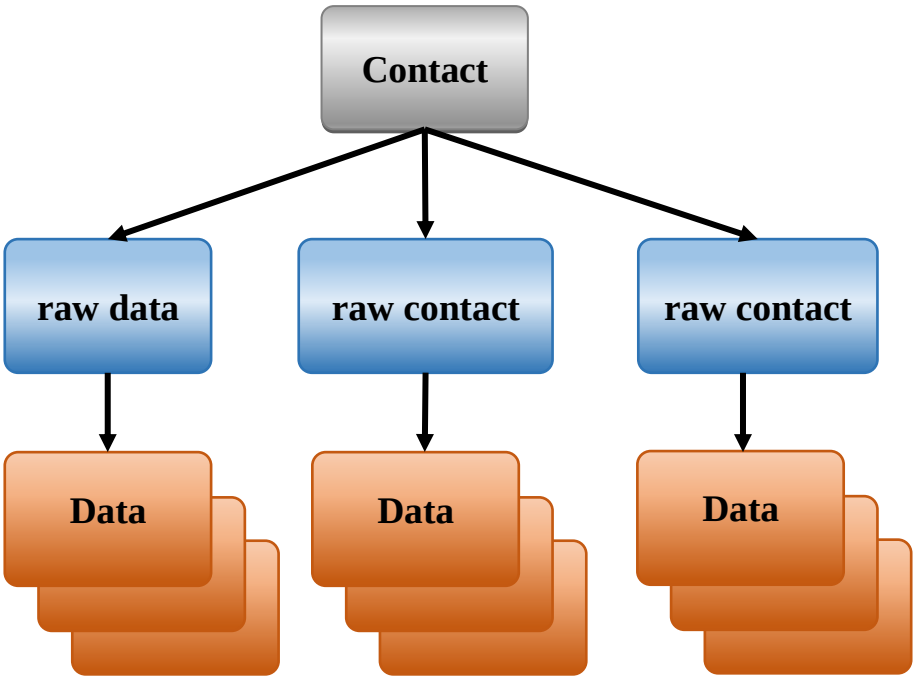

Рисунок 4.2 – Структура таблиці постачальника контактів

Як імена цих трьох таблиць зазвичай використовуються назви відповідних класів-контрактів. Ці класи визначають константи для URI контенту, назв стовпців і значень в 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 відноситься до призначеного для користувача акаунта і його типу.

Таблиця `ContactsContract.Data`

Рядки в цій таблиці містять відомості про необроблені контакти, такі, як адреса ел. пошти або номери телефонів.

Інші таблиці, представлені класами-контрактами в `ContactsContract`, являють собою допоміжні таблиці, які постачальник контактів використовує

для управління своїми операціями або для підтримки певних функцій, наявних в додатку пристрою «Контакти» або в додатках для телефонного зв'язку.

4.5.2. Необроблені контакти

Необроблений контакт являє собою дані про користувача, що надходить з одного акаунта певного типу. Оскільки як джерело даних про користувача в постачальнику контактів може виступати відразу кілька онлайн-служб, то для того ж самого користувача в постачальнику контактів може існувати кілька необроблених контактів. Це також дозволяє користувачеві об'єднувати призначені для користувача дані з декількох акаунтів одного того самого типу.

Більша частина даних необробленого контакту не зберігається в таблиці `ContactsContract.RawContacts`. Замість цього вони зберігаються в одній або декількох рядках в таблиці `ContactsContract.Data`. У кожному рядку даних є стовпець `Data.RAW_CONTACT_ID`, в якому міститься значення `android.provider.BaseColumns#_ID RawContacts._ID` його батьківського рядка `ContactsContract.RawContacts`.

Важливі стовпці необроблених контактів. У табл. 4.12 вказані стовпці таблиці `ContactsContract.RawContacts`, які мають велике значення. Обов'язково ознайомтеся з примітками, наведеними після цієї таблиці.

Таблиця 4.12 – Важливі стовпці необроблених контактів

Назва стовпця	Використання	Примітки
`ACCOUNT_NAME`	Ім'я облікового запису для типу акаунта, який виступає в ролі джерела даних для необробленого контакту. Наприклад, ім'я облікового запису Google є одним з ел. адрес пошти Gmail власника пристрою. Додаткові відомості див. у наступному записі `ACCOUNT_TYPE`	Формат цього імені залежить від типу облікового запису. Це не обов'язково адреса ел. пошти
`ACCOUNT_TYPE`	Тип акаунта, який виступає в ролі джерела даних для необробленого контакту. Наприклад, тип акаунта Google – `com.google.` Завжди вказуйте тип акаунта і ідентифікатор домену, яким ви володієте або керуєте. Це дозволить гарантувати унікальність типу вашого облікового запису	Тип акаунта, який виступає у ролі джерела даних контактів, зазвичай має пов'язаний з ним адаптер синхронізації, який синхронізує дані з постачальником контактів

Продовження табл. 4.12

1	2	3
DELETED	Прапорець deleted для необробленого контакту	Цей прапорець дозволяє постачальнику контактів зберігати рядок всередині себе доти, поки адаптери синхронізації не зможуть видалити цей рядок з серверів, а потім видалити його зі сховищ

Примітки

Наведемо важливі примітки до табл. 4.12 ContactsContract.RawContacts:

- Ім'я необробленого контакту не зберігається в його рядку в таблиці ContactsContract.RawContacts. Замість цього воно зберігається в таблиці ContactsContract.Data у рядку ContactsContract.CommonDataKinds.StructuredName. У необробленого контакту в таблиці ContactsContract.Data є тільки один рядок такого типу.
- **Увага!** Щоб використовувати в рядку необробленого контакту дані власного облікового запису, рядок спочатку необхідно зареєструвати в класі AccountManager. Для цього запропонуйте користувачам додати тип акаунта і його ім'я в список акаунтів. Якщо не зробити цього, постачальник контактів автоматично видалить ваш рядок необробленого контакту.

Наприклад, якщо необхідно, щоб у вашому додатку зберігалися дані контактів для веб-служби в домені com.example.dataservice, і аккаунт користувача служби виглядав таким чином, як becky.sharp@dataservice.example.com, то користувачеві спочатку необхідно додати «тип» акаунта (com.example.dataservice) і його «ім'я» (becky.smart@dataservice.example.com) перш ніж ваш додаток зможе додавати рядки необроблених контактів. Цю вимогу можна вказати в документації для користувачів; також можна запропонувати користувачеві додати тип і ім'я свого облікового запису, або реалізувати обидва ці варіанти.

Джерела даних необроблених контактів. Щоб зрозуміти, що таке необроблений контакт, розглянемо приклад з користувачем Emily Dickinson, на пристрої якої є три наступних акаунти:

– emily.dickinson@gmail.com;

– emilyd@gmail.com;

– *belle_of_amherst* у Twitter.

Вона включила функцію *Синхронізувати контакти* для всіх трьох цих акаунтів в налаштуваннях *Акаунти*.

Припустимо, що Emily Dickinson відкриває браузер, входить в Gmail під ім'ям emily.dickinson@gmail.com, потім відкриває Контакти і додає новий контакт Thomas Higginson. Пізніше вона знову входить в Gmail під ім'ям emilyd@gmail.com і відправляє листа користувачеві Thomas Higginson, який автоматично додається в її контакти. Також вона підписана на новини від *colonel_tom* (акаунт користувача Thomas Higginson в Twitter) в Twitter. В результаті цих дій постачальник контактів створює три необроблених контакти:

1. Необроблений контакт Thomas Higginson, пов'язаний з акаунтом emily.dickinson@gmail.com. Тип цього акаунта – Google.

2. Другий необроблений контакт Thomas Higginson, пов'язаний з акаунтом emilyd@gmail.com. Тип цього акаунта – також Google. Другий необроблений контакт створюється навіть в тому випадку, якщо його ім'я повністю збігається з ім'ям попереднього контакту, оскільки перший був доданий для іншого облікового запису.

3. Третій необроблений контакт Thomas Higginson, пов'язаний з акаунтом belle_of_amherst. Тип цього акаунта – Twitter.

4.5.3. Дані

Як вже зазначалося раніше, дані необробленого контакту зберігаються в рядку ContactsContract.Data, який пов'язаний зі значенням _ID необробленого контакту. Завдяки цьому для одного необробленого контакту може існувати декілька екземплярів того самого типу даних (наприклад, адрес ел. пошти або номерів телефонів). Наприклад, якщо у контакту Thomas Higginson для акаунта emilyd@gmail.com (рядок необробленого контакту Thomas Higginson, пов'язаний з обліковим записом Google emilyd@gmail.com) є адреса ел. пошти thigg@gmail.com і службова адреса ел. пошти thomas.higginson@gmail.com, то постачальник контактів зберігає два рядки адреси ел. пошти і пов'язує їх з цим необробленим контактом.

Зверніть увагу, що в цій таблиці зберігаються дані різних типів. Рядки з відомостями про ім'я, номер телефону, адреса ел. пошти, поштова адреса, фото

і веб-сайт, що відображаються, зберігаються в таблиці ContactsContract.Data. Для спрощення управління цими даними в таблиці ContactsContract.Data передбачені стовпці з описовими іменами, а також інші стовпці з універсальними іменами. Вміст у стовпці з описовим ім'ям має те ж значення, незалежно від типу даних в рядку, тоді як вміст стовпчика з універсальним ім'ям може мати різне значення залежно від типу даних.

Описові імена стовпців. Ось деякі приклади стовпців з описовими іменами:

– RAW_CONTACT_ID. Значення в стовпці _ID необробленого контакту для цих даних.

– MIMETYPE. Тип даних, що зберігаються в цьому рядку, у вигляді типу MIME, що настроюється. 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, в яких містяться основні дані для цього типу. Наприклад, якщо користувач натиснув і утримує номер телефону контакту і вибрав параметр **Використовувати за замовчуванням**, то в стовпці ContactsContract.Data з цим номером телефону у відповідному стовпці IS_PRIMARY задається значення, відмінне від нуля.

Універсальні імена стовпців. Існує 15 загальнодоступних стовпців з універсальними іменами (DATA1–DATA15) і чотири додаткових стовпчики (SYNC1–SYNC4), які використовуються тільки адаптерами синхронізації. Константи стовпців з універсальними іменами застосовуються завжди, незалежно від типу даних, що містяться в стовпці.

Стовпець DATA1 є індексованим. Постачальник контактів завжди використовує цей стовпець для даних, які, як він очікує, будуть цільовими в запитах. Наприклад, в рядку з адресами ел. пошти в цьому стовпці вказується фактична адреса ел. пошти.

Зазвичай стовпець DATA15 зарезервований для даних великих двійкових об'єктів (BLOB), таких, як мініатюри фотографій.

Імена стовпців за типами строк. Для спрощення роботи за допомогою стовпців певного типу рядків у постачальнику контактів також передбачені константи для імен стовпців за типами рядків, які визначені в підкласах класу `ContactsContract.CommonDataKinds`. Ці константи просто привласнюють тому самому імені стовпця різні константи, що дозволяє вам отримувати доступ до даних в рядку певного типу.

Наприклад, клас `ContactsContract.CommonDataKinds.Email` визначає константи імені стовпця для рядка `ContactsContract.Data`, у якому є тип MIME `Email.CONTENT_ITEM_TYPE`. У цьому класі міститься константа `ADDRESS` для стовпця адреси ел. пошти. Фактичне значення `ADDRESS` – data1, яке збігається з універсальним ім'ям стовпця.

Увага! Не додавайте свої дані, що настроюються, в таблицю `ContactsContract.Data` за допомогою рядка, в 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 з типом MIME `Email.CONTENT_ITEM_TYPE`, у якому в стовпці `DATA1` замість адреси електронної пошти міститься ім'я користувача. Якщо вкажете в рядку власний тип MIME, що настроюється, можна вільно вказувати власні імена стовпців за типами рядків і використовувати їх так, як забажаєте.

На рис. 4.3 показано, як стовпці з описовими іменами і стовпці даних відображаються у рядку `ContactsContract.Data`, а також як імена стовпців за типом рядків «накладаються» на універсальні імена стовпців.

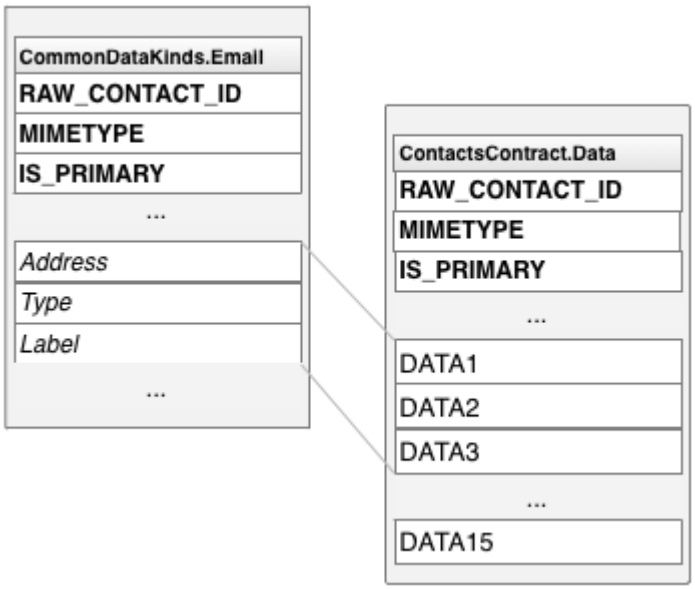

Рисунок 4.3 – Імена стовпців за типами рядків та універсальні імена стовпців

Класи імен стовпців за типами строк. У табл. 4.13 перераховані класи імен стовпців за типами рядків, які частіше за все використовуються.

Таблиця 4.13 – Класи імен стовпців за типами рядків

Клас зіставлення	Тип даних	Примітки
`ContactsContract.CommonDataKinds.StructuredName`	Дані про ім'я необробленого контакту, пов'язаного з цим рядком даних	У необробленого контакту є тільки один рядок такого типу
`ContactsContract.CommonDataKinds.Photo`	Основна фотографія необробленого контакту, пов'язаного з цим рядком даних	У необробленого контакту є тільки один рядок такого типу
`ContactsContract.CommonDataKinds.Email`	Адреса ел. пошти необробленого контакту, пов'язаного з цим рядком даних	У необробленого контакту може бути декілька адрес ел. пошти
`ContactsContract.CommonDataKinds.StructuredPostal`	Поштова адреса необробленого контакту, пов'язаного з цим рядком даних	У необробленого контакту може бути декілька поштових адрес
`ContactsContract.CommonDataKinds.GroupMembership`	Ідентифікатор, за допомогою якого необроблений контакт пов'язаний із однією з груп у постачальнику контактів	Групи є необов'язковим компонентом для типу акаунта і імені акаунта

Контакти. Постачальник контактів об'єднує рядки з необробленими контактами для всіх типів акаунтів і імен акаунтів для створення **контакту**. Це дозволяє спростити відображення і зміну всіх даних, зібраних щодо користувача. Постачальник контактів управляє процесом створення рядків контактів і агрегуванням необроблених контактів, у яких є рядок контакту. Ні додаткам, ні адаптерам синхронізації не дозволяється додавати контакти, а деякі стовпці в рядку контакту доступні тільки для читання.

Примітка. Якщо спробувати додати контакт в постачальник контактів за допомогою методу `insert()`, буде видано виняток

UnsupportedOperationException. Якщо ви спробуєте оновити стовпець, доступний тільки для читання, цю дію буде проігноровано.

При додаванні нового необробленого контакту, який не відповідає жодному з існуючих контактів, постачальник контактів створює новий контакт. Постачальник чинить аналогічно у разі, якщо дані в рядку існуючого необробленого контакту змінюються таким чином, що вони більше не відповідають контакту, з яким вони раніше були пов'язані. При створенні додатком або адаптером синхронізації нового контакту, який відповідає існуючому контакту, новий контакт об'єднується з існуючим контактом.

Постачальник контактів пов'язує рядок контакту з його рядками необробленого контакту за допомогою стовпчика _ID рядка контакту у таблиці Contacts. Стовпець CONTACT_ID в таблиці необроблених контактів ContactsContract.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 виступає у ролі «постійного посилання» 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'язаний з форматом стовпчика android.provider.BaseColumns#_ID.

На рис. 4.4 показано взаємозв'язок цих трьох основних таблиць.

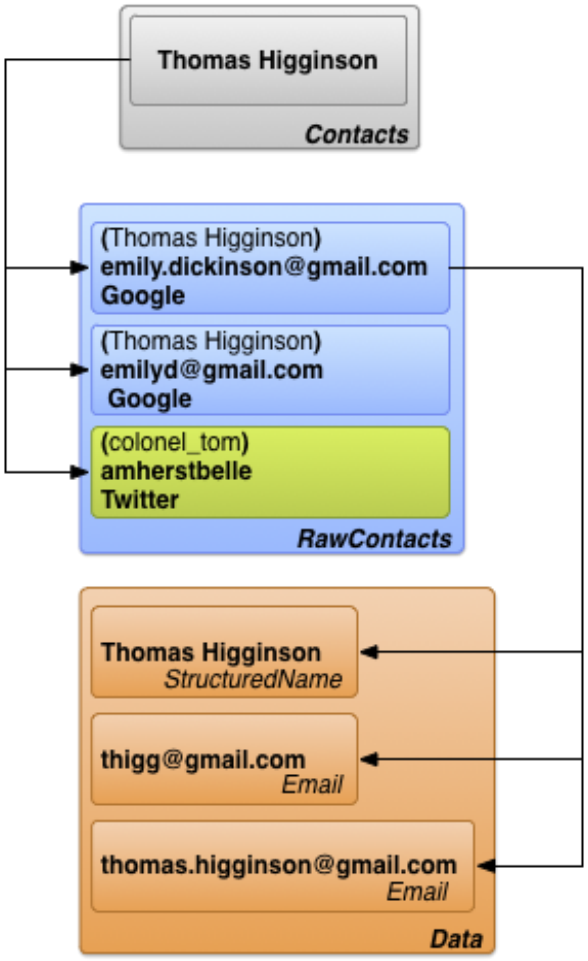

Рисунок 4.4 – Взаємозв’язок між таблицями контактів, необроблених контактів і відомостей

4.5.4. Дані, отримані від адаптера синхронізації

Користувач вводить дані контактів прямо на пристрої, проте дані також надходять в постачальник контактів з 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-служба, з якою працює адаптер синхронізації, визначається за типом акаунта. Кожен адаптер синхронізації працює з одним типом акаунта, проте він може підтримувати кілька імен акаунтів такого типу. Зазначені нижче визначення дозволяють точніше охарактеризувати зв’язок між типами і іменами акаунтів і адаптерами синхронізації і службами.

Тип акаунта. Визначає службу, в якій користувач зберігає дані. У більшості випадків для роботи зі службою користувачеві необхідно пройти перевірку справжності. Наприклад, Google Контакти являє собою тип акаунтів, що

позначається кодом google.com. Це значення відповідає типу акаунта, використовуваного AccountManager.

Ім'я акаунта. Визначає конкретний акаунт або ім'я для входу для типу аккаунта. Акаунти «Контакти Google» – це те саме, що й акаунти Google, в якості імені яких використовується адреса ел. пошти. В інших службах може використовуватися ім'я користувача, що складається з одного слова, або числовий ідентифікатор.

Типи акаунтів не обов'язково повинні бути унікальними. Користувач може створити кілька облікових записів Google Контакти та завантажити дані з них у постачальник контактів; це може статися у разі, якщо у користувача є один набір персональних контактів для особистого облікового запису, а інший набір – для службового акаунта. Імена акаунтів зазвичай унікальні. Разом вони формують певний потік даних між постачальником контактів і зовнішніми службами.

Якщо необхідно передати дані зі служби в постачальник контактів, то треба створити власний адаптер синхронізації.

На рис. 4.5 показано, яку роль виконує постачальник контактів в потоці передачі даних про користувачів. В області, зазначеній на цьому рисунку як *sync adapters*, кожен адаптер промаркований відповідно до підтримуваних ним типом акаунтів.

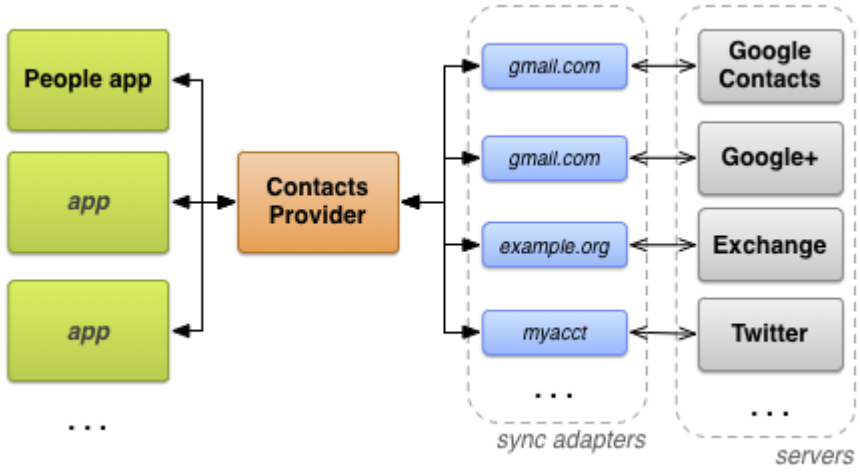

Рисунок 4.5 – Потік передачі даних через постачальника контактів

4.5.5. Необхідні дозволи

Додатки, яким потрібен доступ до постачальника контактів, повинні запросити такі дозволи:

– *Доступ на читання однієї або декількох таблиць*. `READ_CONTACTS`, вказаний у файлі `AndroidManifest.xml` з елементом `<uses-permission>` у

вигляді <uses-permission android:name="android.permission.READ_CONTACTS">.

– *Доступ на запис в одну або кілька таблиць*. WRITE_CONTACTS, вказаний 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.

Пам'ятайте, що дані про контакти користувача є особистими і конфіденційними. Користувачі піклуються про свою конфіденційність, а тому не хочуть, щоб програми збирали дані про них самих або їхні контакти. Якщо користувачеві не до кінця ясно, для чого потрібен дозвіл на доступ до даних контактів, вони можуть привласнити вашому додатку низький рейтинг або взагалі відмовляться його встановлювати.

4.5.6. 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 з контактів користувача. Рядок контактів профілю пов'язаний з рядком необроблених контактів в кожній із систем, в якій використовується профіль. У кожному рядку необробленого контакту профілю може міститися декілька рядків даних. Константи для доступу до профілю користувача подані в класі ContactsContract.Profile.

Для доступу до профілю користувача потрібні особливі дозволи. Крім дозволів READ_CONTACTS та WRITE_CONTACTS, які потрібні для читання і запису, щоб отримати доступ до профілю користувача, необхідні дозволи android.Manifest.permission#READ_PROFILE і android.Manifest.permission#WRITE_PROFILE на читання і запис відповідно.

Завжди слід пам'ятати, що профіль користувача являє собою конфіденційну інформацію. Дозвіл android.Manifest.permission #READ_PROFILE надає вам доступ до особистої інформації на пристрої користувача. В описі свого додатку обов'язково вкажіть, для чого вам потрібен доступ до профілю користувача.

Щоб отримати рядок контакту з профілем користувача, викличте метод ContentResolver.query(). Задайте для URI контенту значення CONTENT_URI і не вказуйте ніяких критеріїв вибірки. Цей URI контенту також можна використовувати як основний URI для отримання необроблених контактів або деталей свого профілю. Нижче подано фрагмент коду для отримання даних профілю:

```
// Встановіть стовпці для вилучення для профілю користувача
mProjection = new String[]
    {
        Profile._ID,
        Profile.DISPLAY_NAME_PRIMARY,
        Profile.LOOKUP_KEY,
        Profile.PHOTO_THUMBNAIL_URI
    };

// Витягує профіль з контактів постачальника
mProfileCursor =
        getContentResolver().query(
                Profile.CONTENT_URI,
                mProjection ,
                null,
                null,
                null);
```

Примітка. Якщо при витяганні кількох рядків контактів необхідно визначити, який з них є профілем користувача, зверніться до колонки `IS_USER_PROFILE` цього рядка. Якщо в цьому стовпці вказано значення «1», то в цьому рядку знаходиться профіль користувача.

4.5.7. Метадані постачальника контактів

Постачальник контактів управляє даними, які використовуються для відстеження стану контактної інформації в репозиторії. Ці метадані про репозиторії зберігаються в різних місцях, включаючи рядки необроблених контактів, дані і рядки таблиці контактів, таблицю `ContactsContract.Settings` і таблицю `ContactsContract.SyncState`. У табл. 4.14 вказано значення кожного з цих фрагментів метаданих.

4.5.8. Доступ до постачальника контактів

У цьому розділі розглядаються інструкції по отриманню доступу до даних з постачальника контактів, зокрема такі:

– запити об’єктів;
– пакетна зміна;
– отримання і 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 ви, можливо, захочете 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, в постачальнику контактів є **об'єкти**, які виступають в ролі з'єднувачів бази даних між таблицями.

Таблиця 4.14 – Метадані в постачальнику контактів

Таблиця	Стовпець	Значення	Опис
1	2	3	4
ContactsContract.RawContacts	DIRTY	«0» – з моменту останньої синхронізації зміни не вносилися	Служить для позначки необроблених контактів, які були змінені на пристрої і які необхідно знову синхронізувати з сервером. Значення встановлюється автоматично постачальником контактів при оновленні рядків додатками Android. Адаптери синхронізації, які вносять зміни в необроблені контакти або таблиці даних, повинні 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 розглядатися як локальні зміни, які підлягають відправленню на сервер, навіть якщо джерелом таких змін був сам сервер
		«1» – з моменту останньої синхронізації були внесені зміни, які необхідно відобразити і на сервері	
ContactsContract.RawContacts	VERSION	Номер версії цього рядка	Постачальник контактів автоматично збільшує це значення при кожній зміні в рядку або в пов'язаних з ним даних
ContactsContract.Data	DATA_VERSION	Номер версії цього рядка	Постачальник контактів автоматично збільшує це значення при кожній зміні в рядку даних
ContactsContract.Groups	GROUP_VISIBLE	«0» – контакти, подані в цій групі, не повинні відображатися в інтерфейсах користувача додатків Android	Цей стовпець призначений для відомостей про сумісність з серверами, що дозволяють користувачам приховувати контакти в певних групах.
		«1» – контакти, подані в цій групі, можуть відображатися в інтерфейсах користувача додатків	

Продовження табл. 4.14

1	2	3	4
`ContactsContract.RawContacts`	`SOURCE_ID`	Рядкове значення, яке є унікальним ідентифікатором даного необробленого контакту в акаунті, в якому він був створений	Коли адаптер синхронізації створює новий необроблений контакт, в цьому стовпці слід вказати унікальний ідентифікатор цього необробленого контакту на сервері. Коли ж додаток Android створює новий необроблений контакт, то у додатку слід залишити цей стовпець порожнім. Це служить сигналом для адаптера синхронізації для створення нового необробленого контакту на сервері і отримання значення `SOURCE_ID`. Зокрема, ідентифікатор джерела повинен бути **унікальним** для кожного типу акаунта і незмінним при синхронізації. Унікальний: у кожного необробленого контакту для акаунта повинен бути свій власний ідентифікатор джерела. Якщо не застосувати цю вимогу, у вас обов'язково виникнуть проблеми з додатком «Контакти». Зверніть увагу, що два необроблених контакти того самого *типу* акаунта можуть мати однаковий ідентифікатор джерела. Наприклад, необроблений контакт Thomas Higginson для акаунта `emily.dickinson@gmail.com` може мати такий же ідентифікатор джерела, що і контакт Thomas Higginson для акаунта `emilyd@gmail.com`. Стабільний: ідентифікатори джерел являють собою 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 повинні бути такими ж, як і раніше. Якщо не застосувати цю вимогу, перестануть працювати ярлики

Закінчення табл. 4.14

1	2	3	4
ContactsContract.SyncState	(усі)	Ця таблиця використовується для зберігання метаданих для вашого адаптера синхронізації	За допомогою цієї таблиці можна на постійній основі зберігати на пристрої відомості про стан синхронізації та інші пов'язані з синхронізацією дані
ContactsContract.Settings	UNGROUPED_VISIBLE	«0» – для цього облікового запису і акаунтів цього типу контакти, які не належать до групи, не відображаються в інтерфейсах користувача додатків Android	За замовчуванням контакти приховані, якщо жоден з їх необроблених контактів не належить групі (належність необробленого контакту до групи вказується в одному або декількох рядках ContactsContract.CommonDataKinds.GroupMembership у таблиці ContactsContract.Data). Встановивши цей прапорець в рядку таблиці ContactsContract.Settings для типу аккаунта і імені аккаунта, можна примусово зробити видимими контакти, які не належать будь-якій групі. Один з варіантів використання цього прапорця – відображення контактів з серверів, на яких не використовуються групи
		«1» – для цього облікового запису і акаунтів цього типу контакти, які не належать до групи, відображаються в інтерфейсах користувача додатків Android	

Об'єкт являє собою подобу таблиці, що складається з обраних стовпців батьківської таблиці і її дочірньої таблиці. При запиті об'єкта надається проекція і критерії пошуку на основі доступних в об'єкті стовпців. В результаті ви отримуєте об'єкт Cursor, в якому міститься один рядок для кожного витягнутого рядка дочірньої таблиці. Наприклад, якщо запросити об'єкт ContactsContract.Contacts.Entity для імені контакту і всі рядки ContactsContract.CommonDataKinds.Email для всіх необроблених контактів, що відповідають цьому імені, ви отримаєте 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 і подальшого запиту до дочірньої таблиці з використанням отриманого

ідентифікатора. Крім того, постачальник контактів обробляє запит об'єкта за одну транзакцію, що гарантує внутрішню узгодженість отриманих даних.

Примітка. Об'єкт зазвичай містить не всі стовпці батьківської і дочірньої таблиць. Якщо ви спробуєте змінити ім'я стовпця, який відсутній в списку констант імені стовпця для об'єкта, то отримаєте `Exception`.

Нижче наведено приклад коду для отримання всіх рядків необробленого контакту для контакту. Це фрагмент більшого додатка, в якому є дві операції: основна і для отримання відомостей. Основна операція служить для отримання списку рядків контактів; при виборі 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, які пов'язані з обраним контактом.

Фрагмент коду операції «для отримання відомостей»:

```
...
    /*
     * Додає шлях суті 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 ідентифікованого за  LOADER_ID.
    getLoaderManager().initLoader(
            LOADER_ID,  // Ідентифікатор завантажувача для
ініціалізації
            null,       // Аргументи для завантажувача (в даному ви-
падку, 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 подання, що
відображають дані
            0);                             // прапори

    // Устанавлює адаптер ListView.
```

```
    mRawContactList.setAdapter(mCursorAdapter);
...
@Override
public Loader<Cursor> onCreateLoader(int id, Bundle args) {

    /*
     * Встановлює стовпці для вилучення.
     * RAW_CONTACT_ID включено для ідентифікації необробленого кон-
такту, пов’язаного з рядком даних.
     * DATA1 містить перший стовпець в рядку даних (як правило,
найважливіший з них).
     * MIMETYPE вказує тип даних в рядку даних.
     */
    String[] projection =
        {
            ContactsContract.Contacts.Entity.RAW_CONTACT_ID,
            ContactsContract.Contacts.Entity.DATA1,
            ContactsContract.Contacts.Entity.MIMETYPE
        };

    /*
     * Проводить сортування витягнутого курсора необробленого ID кон-
такта, щоб зберегти всі рядки даних для одного
     * контакту, що зіставляються один з одним.
     */
    String sortOrder =
            ContactsContract.Contacts.Entity.RAW_CONTACT_ID +
            " ASC";

    /*
     * Повертає 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 і їхні ряди даних
            null,                     //
            sortOrder);               // Сортування по  ID необробле-
ного контакту.
}
```

По завершенні завантаження `LoaderManager` виконує зворотний виклик методу `onLoadFinished()`. Одним із вхідних аргументів для цього методу є `Cursor` з результатом запиту. У своєму додатку можна отримати дані з цього об’єкта `Cursor` для його відображення або подальшої роботи з ним.

Пакетне перетворення. При кожній нагоді дані в постачальнику контактів слід вставляти, оновлювати і 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, ваші зміни завжди знаходитимуться в рамках сховища контактів і завжди будуть узгодженими. Пакетне перетворення також спрощує вставку необробленого контакту одночасно з його детальними даними.

Примітка. Щоб змінити *окремий* необроблений контакт, рекомендується відправити намір в додаток для роботи з контактами на пристрої замість обробки зміни в вашому 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 рамках якомога меншої кількості окремих списків і при цьому запобігти блокуванню роботи системи, слід задати **межі** для однієї або декількох операцій. Межа є об'єктом `ContentProviderOperation`, в якості значення параметра `isYieldAllowed()` якого задано `true`. При досягненні постачальником контактів межі він призупиняє свою роботу, щоб могли виконуватися 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 однієї транзакції. З цієї причини вам 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 і пов'язані з ним рядки даних, або для останньої операції з набором рядків, пов'язаних з одним контактом.

Межі також є одиницями атомарних операцій. Всі операції доступу в проміжку між двома межами завершуються або успіхом, або збоєм в рамках однієї одиниці. Якщо будь-яка з меж не задана, найменшою атомарною операцією вважається весь пакет операцій. Використання меж дозволяє запобігти зниженню продуктивності системи і одночасно забезпечити виконання набору операцій на атомарному рівні.

Зміна зворотніх посилань. При вставлянні нового рядка необробленого контакту і пов'язаних з ним рядів даних у вигляді набору об'єктів `ContentProviderOperation` вам доводиться пов'язувати рядки даних з ряд-

ком необробленого контакту шляхом вставляння значення android.provider.BaseColumns#_ID необробленого контакту у вигляді значення RAW_CONTACT_ID. Однак це значення недоступно, поки створюєте ContentProviderOperation для рядка даних, оскільки ще не застосували ContentProviderOperation для рядка необробленого контакту. Щоб обійти це обмеження, в класі ContentProviderOperation.Builder передбачений метод withValueBackReference(). За допомогою цього методу можна встановлювати або змінювати стовпець з результатом попередньої операції.

У методі withValueBackReference() передбачено два аргументи:

1) key – ключ для пари «ключ-значення». Значенням цього аргументу має бути ім'я стовпця в таблиці, яку змінюєте.

2) previousResult – індекс значення, що починається з «0», в масиві об'єктів ContentProviderResult з методу applyBatch(). По мірі виконання пакетних операцій результат кожної операції зберігається в проміжному масиві результатів. Значенням previousResult є індекс одного з цих результатів, який витягується і зберігається зі значенням key. Завдяки цьому можна вставити новий запис необробленого контакту і отримати назад його значення android.provider.BaseColumns#_ID, а потім створити «зворотне посилання» на значення при додаванні рядка ContactsContract.Data.

Цілком весь результат створюється при першому виклику методу applyBatch(). Розмір результату дорівнює розміру ArrayList наданих вами об'єктів ContentProviderOperation. Однак усім елементам в масиві результатів присвоюється значення null, і при спробі скористатися зворотнbм посиланням на результат операції, що ще не була виконана, метод withValueBackReference() видає Exception.

Нижче наведено фрагменти коду для вставляння нового необробленого контакту і його даних в пакетному режимі. Вони включають код, який задає межа і використовує зворотне посилання. Ці фрагменти яляють собою розширену версію методу createContacEntry(), який входить в клас ContactAdder у прикладі додатка Contact Manager.

Перший фрагмент коду служить для отримання даних контакту з призначеного для користувача інтерфейсу. На цьому етапі користувач уже вибрав акаунт, для якого необхідно додати новий необроблений контакт:

```
// Створює контакт з поточних значень призначеного для користувача ін-
терфейсу, з використанням поточного обраного рахунку.
protected void createContactEntry() {
    /*
     * Отримує значення з призначеного для користувача інтерфейсу
```

```
     */
    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());
```

У наступному фрагменті коду створюється операція для вставляння рядка необробленого контакту в таблицю `ContactsContract.RawContacts`:

```
    /*
     * Підготовлює пакетну обробку для вставки нового необробленого
контакту і його дані. Навіть якщо
     * Постачальник Контактів не має ніяких даних для цієї людини, ви
не можете додати контакт,
     * тільки необроблений контакт. Постачальник Контактів потім до-
дасть контакт автоматично.
     */
     // Створює новий масив об'єктів ContentProviderOperation.
    ArrayList<ContentProviderOperation> ops =
            new ArrayList<ContentProviderOperation>();
    /*
     * Створює новий необроблений контакт з його типом облікового за-
пису (тип сервера) і ім'я облікового запису
     * (рахунок користувача). Пам'ятайте, що псевдонім не зберігається
в цьому рядку, а в
     * рядку даних StructuredName. Інші дані не потрібні.
     */
    ContentProviderOperation.Builder op =

ContentProviderOperation.newInsert(ContactsContract.RawContacts.CONTEN
T_URI)
            .withValue(ContactsContract.RawContacts.ACCOUNT_TYPE,
mSelectedAccount.getType())
            .withValue(ContactsContract.RawContacts.ACCOUNT_NAME,
mSelectedAccount.getName());
    // Будує опреацію і додає її до масиву операцій
    ops.add(op.build());
```

Потім код створює рядки даних для рядків імені, телефону і адреси ел. пошти, що відображуються.

Кожен об'єкт використовує метод `withValueBackReference()` для отримання `RAW_CONTACT_ID`. Посилання повертається назад до об'єкту `ContentProviderResult` з першої операції, в результаті чого додається рядок необробленого контакту і повертається його нове значення

android.provider.BaseColumns#_ID. Після цього кожен рядок даних автоматично пов'язується за своїм RAW_CONTACT_ID з новим рядком ContactsContract.RawContacts, якому він належить.

Об'єкт ContentProviderOperation.Builder, який додає рядок адреси ел. пошти, позначається прапором за допомогою методу withYieldAllowed(), який задає межі:

```
    // Створює ім'я для нового необробленого контакту, як ряд даних
StructuredName.
    op =

ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI)
            /*
             * з ValueBackReference встановлює значення першого аргу-
менту до значення
             * ContentProviderResult, що індексується другим аргумен-
том. В даному конкретному
             * виклику, ID колонки необробленого контакту
StructuredName рядок даних встановлюється
             * на значення результату, що повертається першою
операцією, 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 коротке ім'я рядка даних для імені в інтерфейсі.

.withValue(ContactsContract.CommonDataKinds.StructuredName.DISPLAY_NA
ME, name);

    // Будує операцію і додає її в масив операцій
    ops.add(op.build());

    // Вставляє вказаний телефонний номер і тип як рядок даних теле-
фону
    op =

ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI)
            /*
             * Встановлює 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)

            // Встановлює номер і тип телефону
            .withValue(ContactsContract.CommonDataKinds.Phone.NUMBER,
phone)
            .withValue(ContactsContract.CommonDataKinds.Phone.TYPE,
phoneType);

    // Будує операцію і додає її в масив операцій
    ops.add(op.build());

    // Вставляє вказану адресу електронної пошти та тип як рядок да-
них телефону
    op =

ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI)
            /*
             * Встановлює 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 ел. адреси
            .withValue(ContactsContract.Data.MIMETYPE,

ContactsContract.CommonDataKinds.Email.CONTENT_ITEM_TYPE)

            // Встановлює ел. адресу та тип

.withValue(ContactsContract.CommonDataKinds.Email.ADDRESS, email)
            .withValue(ContactsContract.CommonDataKinds.Email.TYPE,
emailType);

    /*
     * Демонструється вихідна точка. По закінченню цієї вставки,
партія пакетних операцій
     * дасть пріоритет для інших потоків. Використовуйте після кожно-
го набору операцій, які впливають на
     * один контакт, щоб уникнути зниження продуктивності.
     */
    op.withYieldAllowed(true);

    // Будує операцію і додає її в масив операцій
    ops.add(op.build());
```

В останньому фрагменті коду наведено виклик методу `applyBatch()` для вставляння нового необробленого контакту і його рядків даних:

```
    // 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 в пакетному
режимі. 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();
            // Ввод застереження
            Log.e(TAG, "Exception encountered while inserting contact:
" + e);
    }
}
```

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 були бути внесені в цей же час. Якщо виявиться, що є неузгоджена зміна, транзакцію можна відкотити і виконати повторно.

Оптимістичне управління паралелізмом підходить для мобільних пристроїв, де з 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 установлення блокувань або очікування того, коли інші транзакції скасують свої блокування.

Порядок використання оптимістичного управління паралелізмом при оновленні одного рядка `ContactsContract.RawContacts` такий:

1. Вийміть стовпець `VERSION` необробленого контакту, а також інші дані, які необхідно витягти.

2. Створіть об'єкт `ContentProviderOperation.Builder`, відповідний для застосування обмеження, за допомогою методу `newAssertQuery(Uri)`. Для URI контенту використовуйте `RawContacts.CONTENT_URI`, додавши до нього `android.provider.BaseColumns#_ID` необробленого контакту.

3. Для об'єкта `ContentProviderOperation.Builder` викличте метод `withValue()`, щоб порівняти стовпець `VERSION` з номером версії, яку ви тільки що отримали.

4. Для того ж об'єкта `ContentProviderOperation.Builder` викличте метод `withExpectedCount()`, щоб переконатися в тому, що перевірочне твердження перевіряє тільки один рядок.

5. Викличте метод `build()`, щоб створити об'єкт `ContentProviderOperation`, потім додайте цей об'єкт як перший об'єкт в `ArrayList`, що передається в метод `applyBatch()`.

6. Застосуйте пакетну транзакцію.

Якщо в проміжку між зчитуванням рядка і спробою його зміни рядок необробленого контакту був оновлений іншою операцією, assert `ContentProviderOperation` завершиться збоєм, і у виконанні всього пакета операцій буде відмовлено. Можна вибрати повторне виконання пакета або виконати іншу дію.

У прикладі коду демонструється, як створити assert `ContentProviderOperation` після запиту одного необробленого контакту з допомогою `CursorLoader`:

```
/*
 * Додаток використовує CursorLoader для того, щоб запросити таблицю
необроблених контактів. Система викликає цей метод
 * коли закінчиться завантаження.
 */
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {

    // Отримує _ID і VERSION значення необробленого контакту
    mRawContactID =
cursor.getLong(cursor.getColumnIndex(BaseColumns._ID));
    mVersion = cur-
sor.getInt(cursor.getColumnIndex(SyncColumns.VERSION));
}

...

// Встановлює 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

...



// 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
    }
```

Отримання і зміна даних за допомогою намірів. За допомогою відправлення намірів в додаток для роботи з контактами, які записано на пристрій, можна в обхід отримати доступ до постачальника контактів. Намір запускає інтерфейс користувача програми на пристрої, за допомогою якого користувач може працювати з контактами. Такий тип доступу дозволяє користувачеві виконувати наступні дії:

– вибір контактів зі списку та їх повернення в додаток для подальшої роботи;

– зміна даних існуючого контакту;

– вставляння нового необробленого контакту для будь-яких інших акаунтів;

– видалення контакту або його даних.

Якщо користувач вставляє або оновлює дані, можна спочатку зібрати ці дані, а потім відправити їх разом із наміром.

При використанні намірів для доступу до постачальника контактів за допомогою додатка для роботи з контактами, наявного на пристрої, вам не потрібно створювати власний інтерфейс користувача або код для доступу до постачальника. Також вам не потрібно запитувати дозвіл на читання або запис в постачальнику. Додаток для роботи з контактами, наявний на пристрої, може делегувати вам дозвіл на читання контакту, і, оскільки ви вносите зміни в постачальник через інший додаток, вам не потрібен дозвіл на запис.

У табл. 4.15 наведено зведені відомості про операції, тип MIME і значення даних, які використовуються для доступних задач. Значення додаткових даних, які можна використовувати для `putExtra()`, вказані в довідковій документації по `ContactsContract.Intents.Insert`.

Додаток для роботи з контактами, наявний на пристрої, не дозволяє вам видаляти необроблені контакти або будь-які його дані за допомогою намірів. Замість цього для обробки необробленого контакту використовуйте метод `ContentResolver.delete()` або `ContentProviderOperation.newDelete()`.

Нижче наведено фрагмент коду для створення і відправлення намірів, який вставляє новий необроблений контакт і його дані:

```
// Отримує значення з призначеного для користувача інтерфейсу
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 відправки в додаток контактів пристрою
Intent insertIntent = new Intent(ContactsContract.Intents.Insert.ACTION);

// Встановлює тип MIME
insertIntent.setType(ContactsContract.RawContacts.CONTENT_TYPE);

// Встановлює нове ім’я контакту
insertIntent.putExtra(ContactsContract.Intents.Insert.NAME, name);

// Встановлює нову компанію і посаду
insertIntent.putExtra(ContactsContract.Intents.Insert.COMPANY, company);
insertIntent.putExtra(ContactsContract.Intents.Insert.JOB_TITLE,
jobtitle);

/*
 * Демонструє додавання рядків даних в якості списку масиву, пов’язаного
з ключем DATA
 */
```

```
// Визначає список масиву, щоб утримувати об’єкти ContentValues для кож-
ного рядка
ArrayList<ContentValues> contactData = new ArrayList<ContentValues>();

/*
 * Визначає необроблений рядок контакту
 */

// Встановлює рядок в якості об’єкта ContentValues
ContentValues rawContactRow = new ContentValues();

// Додає тип облікового запису та ім’я 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);

/*
 * Встановлює рядок даних телефонного номера
 */

// Встановлює рядок в якості об’єкта ContentValues
ContentValues phoneRow = new ContentValues();

// Вказує тип MIME для цього рядка даних (всі рядки даних повинні бути
відзначені за їх типом)
phoneRow.put(
        ContactsContract.Data.MIMETYPE,
        ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE
);

// Додає номер телефону і його тип до ряду
phoneRow.put(ContactsContract.CommonDataKinds.Phone.NUMBER, phone);

// Додає рядок в масив
contactData.add(phoneRow);

/*
 * Встановлює рядок даних електронної пошти
 */

// Встановлює рядок в якості об’єкта ContentValues
ContentValues emailRow = new ContentValues();

// Вказує тип MIME для цього рядка даних (всі рядки даних повинні бути
відзначені за їх 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 в масив
contactData.add(emailRow);

insertIntent.putParcelableArrayListExtra(ContactsContract.Intents.Insert.
DATA, contactData);

// Розсилає намір для старту додатка на пристрої та активності контакту.
startActivity(insertIntent);
```

Цілісність даних. Оскільки в репозиторії контактів міститься важлива і конфіденційна інформація, яка, як очікує користувач, вірна і актуальна, в постачальнику контактів передбачені чітко визначені правила для забезпечення цілісності даних. При зміні даних контактів ви зобов'язані дотримуватися цих правил. Ось найважливіші з цих правил:

1. Завжди додавайте рядок `ContactsContract.CommonDataKinds.StructuredName` до кожного рядка, що додаєте в `ContactsContract.RawContacts`.

2. Якщо в таблиці `ContactsContract.Data` є рядок `ContactsContract.RawContacts` без рядка `ContactsContract.CommonDataKinds.StructuredName`, то можуть виникнути проблеми під час агрегування.

3. Завжди зв'язуйте нові рядки `ContactsContract.Data` з їх батьківськими рядками `ContactsContract.RawContacts`.

4. Рядок `ContactsContract.Data`, який не пов'язаний з `ContactsContract.RawContacts`, не буде показаний у додатку для роботи з контактами, що є на пристрої, а також може призвести до проблем з адаптерами синхронізації.

Слід змінювати дані тільки тих необроблених контактів, якими ви володієте.

Пам'ятайте про те, що постачальник контактів зазвичай управляє даними з декількох акаунтів різних типів або послуг через Інтернет. Необхідно переконатися в тому, що ваш додаток змінює або видаляє дані тільки для тих рядків, які належать вам, а також в тому, що він вставляє дані з використанням тільки тих типів і імені акаунта, якими ви керуєте.

Завжди використовуйте для центрів, URI контенту, шляхів URI, імен стовпців, типів MIME та значень `TYPE` тільки ті константи, які визначені в класі `ContactsContract` і його підкласах.

Це дозволить уникнути помилок. Якщо яка-небудь константа застаріла, компілятор повідомить про це за допомогою відповідного попередження.

Таблиця 4.15 – Наміри в постачальнику контактів

Задача	Дія	Дані	Тип MIME	Примітки
1	2	3	4	5
Вибір контакту зі списку	`ACTION_PICK`	Одне з нижченаведеного: `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 ел. пошти для необробленого контакту)	Не використовується	Відображення списку необроблених контактів або списку даних необробленого контакту (залежно від наданого URI контенту). Викличте метод `startActivityForResult()`, який повертає URI контенту для вибраного рядка. URI поданий у формі URI контенту таблиці, до якого доданий `LOOKUP_ID`рядка. Додаток для роботи з контактами, встановлений на пристрої, делегує цьому URI контенту дозвіл на читання і запис, які діють протягом усього життєвого циклу вашої операції
Вставка нового необробленого контакту	`Insert.ACTION`	Н/Д	`RawContacts.CONTENT_TYPE`, тип MIME для набору необроблених контактів	Відображення екранау **Додати контакт** в додатку для роботи з контактами, який встановлено на пристрої. Відображаються значення додаткових даних, доданих вами в намір. При відправленні за допомогою методу `startActivityForResult()`URI контенту нового доданого необробленого контакту передається назад в метод зворотного виклику `onActivityResult()` вашої операції в аргументі `Intent` у полі data. Щоб отримати значення, викличте метод `getData()`

Продовження табл. 4.15

1	2	3	4	5
Зміна контакту	`ACTION_EDIT`	`CONTENT_LOOKUP_URI` контакту. Операція редактора дозволить користувачеві змінити будь-які дані, пов’язані з цим контактом	`Contacts.CONTENT_ITEM_TYPE` один контакт.	Відображення екрана редагування контакту в додатку для роботи з контактами. Відображаються значення додаткових даних, доданих вами в намір. Коли користувач натискає на кнопку **Готово** для збереження внесених змін, ваша 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 відображається екран редагування або додавання контакту, і відображаються додаткові дані, передані вами в намір. Якщо ваш додаток відображає такі дані контакту, як адреса ел. пошти або номер телефону, скористайтеся цим наміром, щоб дозволити користувачеві додавати дані до існуючого контакту. **Примітка.** Вам не потрібно відправляти значення імені в додаткові дані свого наміру, оскільки користувач завжди обирає існуюче ім’я або додає нове. Крім того, якщо відправити ім’я, а користувач вирішить внести зміни, додаток для роботи з контактами відобразить відправлене вами ім’я, перезаписуючи попереднє значення. Якщо користувач не помітить цього і збереже зміни, старе значення буде втрачено

Рядки даних, які можна налаштувати. Створивши і використовуючи власні типи MIME, можна вставляти, змінювати, видаляти і витягувати в таблиці `ContactsContract.Data` власні рядки даних. Ваші рядки обмежені використанням стовпчика, який визначений в `ContactsContract.DataColumns`, однак можна порівняти ваші власні імена стовпців за типами рядків з іменами стовпців за замовчуванням. Дані для ваших рядків відображаються в додатку для роботи з контактами, який записано на пристрій, проте їх не вдасться змінити або видалити, а користувачі не зможуть додати додаткові дані. Щоб дозволити користувачам змінювати ваші рядки даних, які можна налаштувати, необхідно реалізувати у вашому додатку операцію редактора.

Для відображення даних, що налаштовуються, вкажіть файл `contacts.xml`, що містить елемент `<ContactsAccountType>` і один або декілька його `<ContactsDataKind>` дочірніх елементів.

4.5.9. Адаптери синхронізації постачальника контактів

Постачальник контактів розроблений спеціально для обробки операцій синхронізації даних контактів між пристроєм і службою в Інтернеті. Завдяки цьому користувачі можуть завантажувати існуючі дані на новий пристрій і відправляти їх в новий обліковий запис. Синхронізація також забезпечує надання користувачеві завжди актуальних відомостей, незалежно від джерела їх додавання і внесених до них змін. Ще одна перевага синхронізації полягає в тому, що дані контактів доступні навіть в тому випадку, якщо пристрій не підключений до мережі.

Незважаючи на безліч різних способів реалізації синхронізації даних, в системі Android підключається платформа синхронізації, яка дозволяє автоматизувати виконання таких завдань:

– перевірка доступності мережі;

– планування і виконання синхронізації на основі уподобань користувача;

– перезапуск зупинених процесів синхронізації.

Для використання платформи надається модуль адаптера синхронізації. Кожен адаптер синхронізації є унікальним для служби і постачальника контенту, проте він здатний працювати з декількома обліковими записами в одній службі. У платформі також передбачена можливість використовувати кілька адаптерів синхронізації для тієї ж самої служби або постачальника.

Класи і файли адаптера синхронізації. Адаптер синхронізації реалізується як підклас класу `AbstractThreadedSyncAdapter` і встановлюється в складі

додатка Android. Система дізнається про наявність адаптера синхронізації з елементів в маніфесті вашого додатка, а також з особливого файлу XML, на який є вказівка в маніфесті. У файлі XML визначаються тип акаунта в онлайн-службі і центр постачальника контенту, які разом служать унікальними ідентифікаторами адаптера. Адаптер синхронізації знаходиться в неактивному стані доти, поки користувач не додасть тип акаунта і не включить синхронізацію з постачальником контенту. На цьому етапі система вступає в управління адаптером, за необхідності викликаючи його для синхронізації даних між постачальником контенту і сервером.

Примітка. Використання типу акаунта для ідентифікації адаптера синхронізації дозволяє системі виявляти і групувати адаптери синхронізації, які звертаються до різних служб з тієї ж самої організації. Наприклад, у всіх адаптерів синхронізації для онлайн-служб Google той самий тип аккаунта – `com.google`. Коли користувачі додають на свої пристрої облікового запису Google, всі встановлені адаптери синхронізації групуються разом; кожен з адаптерів синхронізується 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 аутентифікації, яка аналогічна платформі адаптера синхронізації і часто використовується разом з нею. Платформа аутентифікації використовує спільні структури перевірки автентичності, які являють собою підкласи класу `AbstractAccountAuthenticator`. Така структура перевіряє справжність користувача таким чином:

1. Спочатку збираються відомості про ім’я користувача і його пароль або аналогічна інформація (облікові дані користувача).

2. Потім облікові дані оговтуються в 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 аутентифікації, які підтримує структура перевірки автентичності і які вона вирішує надати, наприклад, до маркерів аутентифікації OAuth2.

Незважаючи на те що аутентифікація не вимагається, вона використовується більшістю служб для роботи з контактами. Проте вам не обов’язково використовувати платформу аутентифікації Android для перевірки автентичності.

Реалізація адаптера синхронізації. Щоб реалізувати адаптер синхронізації для постачальника контактів, спочатку треба створити додаток Android, який містить такі компоненти:

– Компонент `Service`, Який відповідає на запити системи для прив'язки до адаптера синхронізації. Коли системі потрібно запустити синхронізацію, вона викликає метод `onBind()` служби, щоб отримати об'єктIBinderдля адаптера синхронізації. Завдяки цьому система може викликати методи адаптера між процесами.

У прикладі адаптера синхронізації ім'ям класу цієї служби є `com.example.android.samplesync.syncadapter.SyncService`.

– Безпосередньо адаптер синхронізації, реалізований як конкретний підклас класу `AbstractThreadedSyncAdapter`. Цей клас не підходить для завантаження даних з сервера, відправки даних з пристрою і вирішення конфліктів. Основну свою роботу адаптер виконує в методі `onPerformSync()`. Цей клас допускає створення тільки одного примірника.

У прикладі адаптера синхронізації адаптер визначається в класі `com.example.android.samplesync.syncadapter.SyncAdapter`.

Підклас класу `Application` виступає в ролі фабрики для єдиного екземпляра адаптера синхронізації. Скористайтеся методом `onCreate()`, щоб створити екземпляр адаптера синхронізації, а потім надайте статичний метод get, щоб повернути єдиний екземпляр в метод `onBind()` служби адаптера синхронізації.

– **Не обов'язково:** компонент `Service`, який відповідає на запити системи для аутентифікації користувачів. `AccountManager` запускає службу, щоб почати процес аутентифікації. Метод `onCreate()` служби створює екземпляр об'єкта структури перевірки автентичності. Коли системі потрібно запустити аутентифікацію акаунта 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 ім'ям класу цієї служби є `com.example.android.samplesync.authenticator.AuthenticationService`.

– **Не обов'язково:** конкретний підклас класу `AbstractAccountAuthenticator`, який обробляє запити на аутентифікацію. У цьому класі є методи, які `AccountManager` викликає для перевірки автентичності облікових даних користувача на сервері. Подробиці процесу аутентифікації значно різняться залежно від технології, що використовується на сервері. Щоб

дізнатися докладніше про аутентифікацію, зверніться до відповідної документації програмного забезпечення використовуваного сервера.

У прикладі адаптера синхронізації структура перевірки автентичності визначається в класі – `com.example.android.samplesync.authenticator.Authenticator`.

– Файли XML, в яких визначаються адаптер синхронізації і структура перевірки автентичності для системи.

Описані раніше компоненти служби адаптера синхронізації і структури перевірки автентичності визначаються в елементах `<service>` в маніфесті додатка. Ці елементи включають дочірні елементи `<meta-data>`, в яких є певні дані для системи.

Елемент `<meta-data>` для служби адаптера синхронізації вказує на файл XML `res/xml/syncadapter.xml`. У свою чергу, в цьому файлі задається URI веб-служби для синхронізації з постачальником контактів, а також тип акаунта 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, в цьому файлі задається тип аккаунта, який підтримує структура перевірки автентичності, а також ресурси для користувача інтерфейсу, які відображаються в процесі аутентифікації. Тип акаунта, зазначений в цьому елементі, повинен збігатися з типом акаунта, який заданий для адаптера синхронізації.

4.5.10. Потоки даних із соціальних мереж

Для управління вхідними даними з соціальних мереж використовуються таблиці `android.provider.ContactsContract.StreamItemsandroid.provider.ContactsContract.StreamItemPhotos`. Можна створити адаптер синхронізації, який додає потік даних з вашої власної мережі в ці таблиці, або можна зчитувати потік даних з цих таблиць і відображати їх у власному додатку. Можна також реалізувати обидва ці способи. За допомогою цих функцій можна інтегрувати служби соціальних мереж в компоненти Android для роботи з соціальними мережами.

Текст з потоку даних із соціальних мереж. Елементи потоку завжди асоціюються з необробленим контактом. Ідентифікатор `android.provider.ContactsContract.StreamItemsColumns#RAW_CONTACT_ID` зв'язується зі значенням `_ID` необробленого контакту. Тип акаунта і його ім'я для необробленого контакту також зберігаються в рядку елемента потоку.

Дані з потоку слід зберігати в таких стовпцях:

– **Android.provider.ContactsContract.StreamItemsColumns#ACCOUNT_TYPE**

Обов'язковий. Тип аккаунта користувача для необробленого контакту, пов'язаного з цим елементом потоку. Не забудьте поставити це значення при вставлянні елемента потоку.

– **android.provider.ContactsContract.StreamItemsColumns#ACCOUNT_NAME**

Обов'язковий. Ім'я облікового запису користувача для необробленого контакту, пов'язаного з цим елементом потоку. Не забудьте поставити це значення при всталянні елемента потоку.

Стовпці ідентифікатора.

Обов'язковий. При вставлянні елемента потоку необхідно вставити такі стовпці ідентифікатора:

- **android.provider.ContactsContract.StreamItemsColumns#CONTACT_ID**: значення `android.provider.BaseColumns#_ID` для контакту, з яким асоційований цей елемент потоку;
- **android.provider.ContactsContract.StreamItemsColumns#CONTACT_LOOKUP_KEY**: значення `android.provider.ContactsContract.ContactsColumns # LOOKUP_KEY` для контакту, з яким асоційований цей елемент потоку;
- **android.provider.ContactsContract.StreamItemsColumns#RAW_CONTACT_ID**: значення `android.provider.BaseColumns # _ID` для необробленого контакту, з яким асоційований цей елемент потоку.

– **android.provider.ContactsContract.StreamItemsColumns#COMMENTS**

Необов'язковий. У ньому зберігається зведена інформація, яку можна відобразити на початку елемента потоку.

– **android.provider.ContactsContract.StreamItemsColumns # TEXT**

Текст елемента потоку: або контент, опублікований джерелом елемента, або опис деякої дії, згенерованої елементом потоку. У цьому стовпці може міститися будь-яке форматування і вбудовані зображення ресурсів, рендеринг яких можна виконати за допомогою методу `fromHtml()`. Постачальник може обрізати занадто довгий контент або замінити його частину трьома крапками, однак він спробує уникнути порушення тегів.

– **android.provider.ContactsContract.StreamItemsColumns#TIMESTAMP**

Текстовий рядок з інформацією про час вставки або поновлення елемента в мілісекундах від початку відліку часу. Обслуговуванням цієї шпальти займаються додатки, які вставляють або оновлюють елементи потоку; постачальник контактів не виконує це автоматично.

Для відображення ідентифікаційної інформації для елементів потоку скористайтеся

```
android.provider.ContactsContract.StreamItemsColumns#RES_ICON,
android.provider.ContactsContract.StreamItemsColumns#RES_LABEL,
android.provider.ContactsContract.StreamItemsColumns#RES_PACKAGE
```

для зв'язування з ресурсами в вашому додатку.

У таблиці android.provider.ContactsContract.StreamItems також є стовпці android.provider.CotactsContract.StreamItemsColumns#SYNC1, android.provider.ContactsContract.StreamItemsColumns#SYNC4, які призначені виключно для адаптерів синхронізації.

Фотографії з потоку даних із соціальних мереж. Фотографії, пов'язані з елементом потоку, зберігаються в таблиці android.provider.ContactsContract.StreamItemPhotos. Стовпець android.provider.ContactsContract.StreamItemPhotosColumns#STREAM_ITEM_ID в цій таблиці пов'язаний зі стовпцем android.provider.BaseColumns #_ID в таблиці android.provider.ContactsContract.StreamItems. Посилання на фотографії зберігаються в таких стовпчиках таблиці:

– android.provider.ContactsContract.StreamItemPhotos # PHOTO (об'єкт BLOB). Подання фотографії в довічному форматі і зі зміненим постачальником розміром для її зберігання і відображення. Цей стовпець доступний для забезпечення зворотної сумісності з попередніми версіями постачальника контактів, які використовувалися для зберігання фотографій. Однак в поточній версії постачальника ми не рекомендуємо використовувати цей стовпець для зберігання фотографій. Замість цього скористайтеся стовпцем android.provider.ContactsContract.StreamItemPhotosColumns#PHOTO_FILE_ID або android.provider.ContactsContract.StreamItemPhotosColumns#PHOTO_URI (або обома стовпцями, як описано далі) для зберігання фотографій в файлі. У цьому стовпці тепер зберігаються мініатюри фотографій, доступних для читання:

– android.provider.ContactsContract.StreamItemPhotosColumns # PHOTO_FILE_ID. Числовий ідентифікатор фотографії для необробленого контакту. Додайте це значення до константи DisplayPhoto.CONTENT_URI, щоб отримати URI контенту для одного файлу фотографії, а потім викличте метод openAssetFileDescriptor (), щоб отримати засіб обробки файлу фотографії.
android.provider.ContactsContract.StreamItemPhotosColumns # PHOTO_URI

URI контент, який вказує на файл фотографії, викличе метод openAssetFileDescriptor(), передавши в нього цей URI, щоб отримати оброблений файл фотографії.

– *Використання таблиць з потоку даних із соціальних мереж.* Ці таблиці працюють аналогічно іншим основним таблицями в постачальнику контактів, за винятком зазначених нижче моментів:

– робота з цими таблицями потребує більше дозволів на доступ. Для читання даних з них вашому додатку має бути наданий дозвіл `android.Manifest.permission # READ_SOCIAL_STREAM`. Для зміни таблиць ваш додаток повинен мати дозвіл `android.Manifest.permission#WRITE_SOCIAL_STREAM`;

– для `android.provider.ContactsContract.StreamItems` таблиці існує обмеження на кількість рядків, яке можна зберігати для кожного необробленого контакту. Після досягнення цього обмеження постачальник контактів звільняє місце для нових рядків елементів потоку шляхом автоматичного видалення рядків з найстарішою міткою `android.provider.ContactsContract.StreamItemsColumns#TIMESTAMP`. Щоб отримати це обмеження, запросіть URI контенту `android.provider.ContactsContract.StreamItems#CONTENT_LIMIT_URI`. Для всіх аргументів, відмінних від URI контенту, можна залишити значення `null`. Запит повертає об'єкт Cursor, в якому міститься один рядок з одним стовпцем `android.provider.ContactsContract.StreamItems #MAX_ITEMS`.

Клас `android.provider.ContactsContract.StreamItems.StreamItemPhotos` визначає дочірню таблицю об'єктів `android.provider.ContactsContract.StreamItemPhotos`, в якій містяться рядки для одного елемента потоку.

Взаємодія з потоками даних із соціальних мереж. Управління потоком даних із соціальних мереж здійснюється постачальником контактів спільно з додатком для управління контактами, наявними на пристрої. Такий підхід дозволяє організувати ефективне використання даних із соціальних мереж з даними про існуючі контакти і має такі функції:

1. Організувавши синхронізацію даних з соціальної служби з постачальником контактів за допомогою адаптера синхронізації, можна отримувати дані про недавню активність контактів користувача і зберігати такі дані в таблицях `android.provider.ContactsContract.StreamItems` і `android.provider.ContactsContract.StreamItemPhotos` для використання в подальшому.

2. Крім регулярної синхронізації, адаптер синхронізації можна налаштувати на отримання додаткових даних при виборі користувачем контакту для перегляду. Завдяки цьому ваш адаптер синхронізації може отримувати фотографії високої роздільної здатності та найактуальніші елементи потоку для контакту.

3. Реєструючи повідомлення в додатку для роботи з контактами і в постачальнику контактів, можна получать наміри при перегляді контакту і оновлювати на цьому етапі дані про стан контакту з вашої служби. Такий підхід може забезпечити більшу швидкодію і менший обсяг використання смуги пропускання, ніж виконання повної синхронізації за допомогою адаптера синхронізації.

4. Користувачі можуть додати контакт у вашу службу соціальної мережі, звернувшись до контакту в додатку, який записано на пристрій. Це реалізується за допомогою функції «запросити контакт», для включення якої використовується поєднання операції, яка додає існуючий контакт в вашу мережу, і файлу XML, в якому наведено відомості про ваш додаток для постачальника контактів і додатки для роботи з контактами.

Регулярна синхронізація елементів потоку за допомогою постачальника контактів виконується так само, як і будь-яка інша синхронізація. Реєстрація повідомлень та запрошення контактів розглядаються в наступних двох розділах.

Реєстрація для обробки переглядів контактів в соціальних мережах. Щоб зареєструвати адаптер синхронізації для отримання повідомлень про перегляди користувачами контакту, управління яким здійснюється вашим адаптером синхронізації, виконайте такі дії:

1. У каталозі `res/xml/` свого проекту створіть файл `contacts.xml`. Якщо у вас вже є цей файл, переходите до наступного кроку.

2. У цьому файлі додайте об’єкт `<ContactsAccountType xmlns: android = "http://schemas.android.com/apk/res/android">`. Якщо цей елемент уже існує, можете переходити до наступного кроку.

3. Щоб зареєструвати службу, якій надсилається повідомлення при відкритті користувачем сторінки з відомостями про контакт в додатку для роботи з контактами, який записано на пристрій, додайте атрибут `viewContactNotifyService = "serviceclass"` до елемента, де `serviceclass` – це повне ім’я класу служби, яка повинна отримати намір з програми для роботи з контактами. Для служби-повідомника використовуйте клас, який є розширенням класу `IntentService`, щоб дозволити службі отримувати наміри. Дані у вхідному намірі містять URI контенту необробленого контакту, обраного користувачем. У службі-повідомнику можна прив’язати адаптер синхронізації, а потім викликати його для поновлення даних для необробленого контакту.

Щоб зареєструвати операцію, яку слід викликати при виборі користувачем елемента потоку або фотографії (або обох елементів), виконайте такі дії:

1. У каталозі res/xml/ свого проекту створіть файл contacts.xml. Якщо у вас вже є цей файл, перейдіть до наступного кроку.

2. У цьому файлі додайте об'єкт <ContactsAccountType xmlns: android = "http://schemas.android.com/apk/res/android">. Якщо цей елемент уже існує, можна переходити до наступного кроку.

3. Щоб зареєструвати одну з ваших операцій для обробки вибору користувачем елемента потоку в додатку, який записано на пристрій, додайте атрибут viewStreamItemActivity = "activityclass" до елемента, де activityclass – це повне ім'я класу операції, яка повинна отримати намір з програми для роботи з контактами.

4. Щоб зареєструвати одну з ваших операцій для обробки вибору користувачем фотографії в потоці в додатку, який записано на пристрій, додайте атрибут viewStreamItemPhotoActivity = "activityclass" до елемента, де activityclass – це повне ім'я класу операції, яка повинна отримати намір з програми для роботи з контактами.

Дані у вхідному намірі містять URI контенту елемента або фотографії, обраних користувачем. Щоб використовувати різні операції для текстових елементів і фотографій, використовуйте обидва атрибути в одному файлі.

Взаємодія зі службою соціальної мережі. Користувачам не обов'язково виходити з додатку, який записаний на пристрій, щоб запросити контакт на сайт соціальної мережі. Замість цього додаток для роботи з контактами може відправити намір для запрошення контакту в одну з ваших операцій. Для цього виконайте такі дії:

1. У каталозі res/xml/ свого проекту створіть файл contacts.xml. Якщо у вас вже є цей файл, перейдыть до наступного кроку.

2. У цьому файлі додайте об'єкт <ContactsAccountType xmlns: android = "http://schemas.android.com/apk/res/android">. Якщо цей елемент уже існує, можна переходити до наступного кроку.

3. Додайте такі атрибути:

– inviteContactActivity = "activityclass";

– inviteContactActionLabel = "@ string/invite_action_label".

Значення activityclass являє собою повне ім'я класу операції, яка повинна отримати намір. Значення invite_action_label – це текстовий рядок, який відображається в меню «Додати підключення» в додатку для роботи з контактами.

Примітка. ContactsSource – це застаріле ім'я тега для ContactsAccountType, яке більше не використовується.

Посилання contacts.xml. У файлі `contacts.xml` містяться елементи XML, які керують взаємодією вашого адаптера синхронізації і вашої програми з постачальником контактів і додатком для роботи з контактами. Ці елементи описані в наступних розділах.

Елемент <`ContactsAccountType`> управляє взаємодією вашої програми з додатком для роботи з контактами. Нижче наведено його синтаксис.

```
<ContactsAccountType
        xmlns: android = "http://schemas.android.com/apk/res/android"
        inviteContactActivity = "activity_name"
        inviteContactActionLabel = "invite_command_text"
        viewContactNotifyService = "view_notify_service"
        viewGroupActivity = "group_view_activity"
        viewGroupActionLabel = "group_action_text"
        viewStreamItemActivity = "viewstream_activity_name"
        viewStreamItemPhotoActivity = "viewphotostream_activity_name">
```

Елемент знаходиться в `res/xml/contacts.xml`, і може містити `<ContactsDataKind>`

Опис: цей елемент оголошує компоненти і елементи призначеного для користувача інтерфейсу, за допомогою яких користувачі можуть запрошувати свої контакти в соціальну мережу, повідомляти користувачів при оновленні одного з їх потоків даних з соціальних мереж та ін.

Зверніть увагу, що префікс атрибута `android:` необов'язково використовувати для атрибутів `<ContactsAccountType>`.

Атрибути:

`inviteContactActivity`

Повне ім'я класу операції в вашому додатку, який необхідно активувати при виборі користувачем елемента «Додати підключення» в додатку для роботи з контактами, який записано на пристрій.

`inviteContactActionLabel`

Текстовий рядок, який відображається для операції, заданої в `inviteContactActivity`, в меню «Додати підключення», наприклад, можна вказати фразу «Слідкуйте за новинами в моїй мережі». Для цього елемента можна використовувати ідентифікатор рядкового ресурсу.

`viewContactNotifyService`

Повне ім'я класу служби у вашому додатку, яка повинна отримувати повідомлення при перегляді контакту користувачем. Таке повідомлення надсилається додатком, який записано на пристрій; завдяки цьому ваш додаток може відкласти виконання операцій, що вимагають обробки великого обсягу да-

них, доти, поки це не буде потрібно. Наприклад, ваш додаток може реагувати на таке повідомлення шляхом зчитування і відображення фотографії контакту з високою роздільною здатністю і найактуальніших елементів потоку даних з соціальної мережі. Приклад служби повідомлень наведено у файлі `NotifierService.java`.

`viewGroupActivity`

Повне ім'я класу операцій, яка може відобразити інформацію про групу, при натисканні користувачем на мітку групи.

`viewGroupActionLabel`

Мітка, яка відображається додатком для елемента користувацького інтерфейсу, за допомогою якої користувач може переглянути групи в вашому додатку.

Наприклад, якщо встановити додаток Google+ на ваш пристрій і виконати синхронізацію даних в Google+ з додатком для роботи з контактами, то кола Google+ будуть позначені в додатку для роботи з контактами як групи на вкладці «Групи». При натисканні на коло Google+ учасники кола відобразяться як група контактів. У верхній частині екрану знаходиться значок Google+; якщо натиснути на нього, управління перейде в додаток Google+. У додатку для управління контактами це реалізовано за допомогою `viewGroupActivity`, в якій значок Google+ використовується як значення `viewGroupActionLabel`.

Для цього атрибута можна використовувати ідентифікатор рядкового ресурсу:

`viewStreamItemActivity`

Повне ім'я класу операції в вашому додатку, який запускає додаток для роботи з контактами, коли користувач вибирає елемент потоку для необробленого контакту.

`viewStreamItemPhotoActivity`

Повне ім'я класу операції в вашому додатку, яку запускає додаток для роботи з контактами, коли користувач вибирає фотографію в елементі потоку для необробленого контакту.

Елемент <*ContactsDataKind*>. Елемент `<ContactsDataKind>` управляє відображенням рядків даних вашого додатка, які налаштовуються, в інтерфейсі додатка для роботи з контактами, який записано на пристрій. Нижче наведено його синтаксис.

```
<ContactsDataKind
        android: mimeType = "MIMEtype"
        android: icon = "icon_resources"
        android: summaryColumn="Column_name"
android: detailColumn = "column_name">
```

Елемент знаходиться в <ContactsAccountType>.

Опис: використовуйте цей елемент для відображення вмісту рядків даних, що налаштовуються, в додатку для роботи з контактами як частини відомостей про необроблений контакт. Кожен дочірній елемент <ContactsDataKind> елемента <ContactsAccountType> являє собою тип рядка даних, що налаштовується, який адаптер синхронізації додає в таблицю ContactsContract.Data. Для кожного використовуваного вами типу MIME, що налаштовується, необхідно додати один елемент <ContactsDataKind>. Вам не потрібно додавати елемент, якщо у вас є рядок даних, що налаштовується, для якого не потрібно відображати дані.

Атрибути:

android: mimeType

Певні типи MIME, які ви налаштовуєте, для одного з ваших типів рядків даних в таблиці ContactsContract.Data. Наприклад, значення vnd.android.cursor.item/vnd.example.locationstatus може бути налаштованим типом MIME для рядка даних, в якій знаходяться записи про останнє відоме місцезнаходження контакту.

android: icon

Графічний ресурс Android, який додаток для роботи з контактами відображає поруч з вашими даними. Використовуйте його для позначення того, що ці дані отримані з вашої служби.

android: summaryColumn

Ім’я стовпця для першого з двох значень, отриманих з рядка даних. Значення відображається у вигляді першого рядка запису в цьому рядку даних. Перший рядок призначений для використання як зведені дані, однак він не обов’язковий. Див. також android:detailColumn.

android: detailColumn

Ім’я стовпця для другого з двох значень, отриманих з рядка даних. Значення відображається у вигляді другого рядка запису в цьому рядку даних. Див. також android: summaryColumn.

4.5.11. Додаткові можливості постачальника контактів

Крім основних функцій, описаних вище, в постачальника контактів передбачені такі корисні функції для роботи з даними контактів:

– групи контактів;

– функції роботи з фотографіями.

Групи контактів. Постачальник контактів може додатково зазначити колекції пов'язаних контактів з даними про групу. Якщо серверу, який пов'язаний з обліковим записом користувача, потрібно зберегти групи, адаптеру синхронізації для типу цього акаунта слід передати дані про групи з постачальника контактів на сервер. При додаванні користувачем нового контакту на сервер і подальшому приміщенні цього контакту в нову групу адаптер синхронізації повинен додати цю нову групу в таблицю `ContactsContract.Groups`. Група або групи, в які входить необроблений контакт, зберігаються в таблиці `ContactsContract.Data` з використанням типу MIME `ContactsContract.CommonDataKinds.GroupMembership`.

Якщо необхідно створити адаптер синхронізації, який буде додавати дані необробленого контакту з сервера в постачальник контактів, а не використовувати групи, то вам необхідно вказати для постачальника, щоб він зробив ваші дані видимими. У коді, який виконується при додаванні користувачем акаунта на пристрій, поновіть рядок `ContactsContract.Settings`, яку постачальник контактів додає для цього облікового запису. У цьому рядку вкажіть в стовпці `Settings.UNGROUPED_VISIBLE` значення «1». Після цього постачальник контактів завжди буде робити ваші дані видимими, навіть якщо ви не використовуєте групи.

Фотографії контактів. У таблиці `ContactsContract.Data` зберігаються фотографії у вигляді рядків `Photo.CONTENT_ITEM_TYPE` типу MIME. Стовпець `CONTACT_ID` в рядку пов'язаний зі стовпцем `android.provider.BaseColumns#_ID` необробленого контакту, якому він належить. Клас `ContactsContract.Contacts.Photo` визначає вкладену таблицю `ContactsContract.Contacts`, у якій міститься інформація про основну фотографію контакту (яка є основною фотографією основного необробленого контакту цього контакту). Аналогічним чином клас `ContactsContract.RawContacts.DisplayPhoto` визначає вкладену таблицю `ContactsContract.RawContacts`, у якій міститься інформація про основну фотографію необробленого контакту.

У довідковій документації по `ContactsContract.Contacts.Photo` і `ContactsContract.RawContacts.DisplayPhoto` містяться приклади отримання інформації про фотографії. На жаль, відсутній клас для зручного витягання мініатюри основної фотографії необробленого контакту, проте можна відправити запит в таблицю `ContactsContract.Data`, вибрати `android.provider.BaseColumns#_ID` необробленого контакту,

Photo.CONTENT_ITEM_TYPEі стовпець IS_PRIMARY, щоб знайти рядок основної фотографії необробленого контакту.

Потоки даних із соціальних мереж також можуть включати фотографії. Вони знаходяться в таблиці android.provider.ContactsContract.StreamItemPhotos.

4.6. Платформа доступу до сховища (Storage Access Framework)

Платформа доступу до сховища (Storage Access Framework, SAF) вперше з'явилася в Android версії 4.4 (API рівня 19). Платформа SAF полегшує користувачам пошук і відкриття документів, зображень та інших файлів у сховищах всіх постачальників, з 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 з SAF за допомогою всього декількох рядків коду.

Платформа SAF включає в себе такі компоненти:

1. Постачальник документів – постачальник контенту, що дозволяє службі зберігання (наприклад, Диск Google) показувати файли, якими він управляє. Постачальник документів реалізується як підклас класу DocumentsProvider. Його схема заснована на традиційній файлової ієрархії, однак фізичний спосіб зберігання даних в постачальнику документів залишається на розсуд розробника. Платформа Android включає в себе кілька вбудованих постачальників документів, таких, як «Завантаження», «Зображення» і «Відео».

2. Клієнтська програма – користувацький додаток, що викликає намір ACTION_OPEN_DOCUMENT і/або ACTION_CREATE_DOCUMENT і приймає файли, які повертаються постачальниками документів.

3. Елемент вибору – системний елемент, що забезпечує користувачам доступ до документів у всіх постачальників документів, які задовольняють критерії пошуку, заданим в клієнтському додатку.

Платформа SAF серед інших надає такі функції:

– дозволяє користувачам шукати контент у всіх постачальників документів, а не тільки у однієї програми;

– забезпечує додатку можливість довготривалого, постійного доступу до документів, що належить постачальнику документів. Завдяки такому доступу

користувачі можуть додавати, редагувати, зберігати і видаляти файли, що зберігаються у постачальника;

– підтримує декілька облікових записів і тимчасові кореневі каталоги, наприклад постачальники на USB-накопичувачах, які з'являються, тільки коли накопичувач вставлений в порт.

4.6.1. Огляд

У центрі платформи SAF знаходиться постачальник контенту, який є підкласом класу `DocumentsProvider`. Усередині постачальника документів дані мають структуру традиційної файлової ієрархії (рис. 4.6):

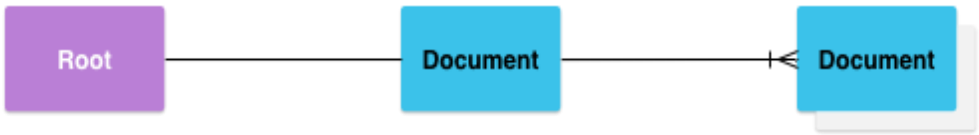

Рисунок 4.6 – Модель даних постачальника документів

На цьому рисунку **Root** (кореневий каталог) вказує на один об'єкт **Document** (документ), який потім розгалужується в ціле дерево.

Зверніть увагу на таке:

- Кожен постачальник документів надає один або кілька «кореневих каталогів», що є відправними точками при обході дерева документів. Кожен кореневий каталог має унікальний ідентифікатор `COLUMN_ROOT_ID` і вказує на документ (каталог), який подає вміст на рівні, що нижчий від кореневого. 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 і вийти з неї.
- У кожному кореневому каталозі знаходиться один документ. Цей документ вказує на кількість документів N, кожен з яких, в свою чергу, може вказувати на один або N документів.
- Кожен сервер сховища показує окремі файли і каталоги, посилаючись на них за допомогою унікального ідентифікатора `COLUMN_DOCUMENT_ID`. Ідентифікатори документів повинні бути унікальними і не змінюватися після присвоєння, 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 (з MIME-типом `MIME_TYPE_DIR`).

• Кожен документ може мати різні властивості, описувані прапорами `COLUMN_FLAGS`, такими, як `FLAG_SUPPORTS_WRITE`, `FLAG_SUPPORTS_DELETE` і `FLAG_SUPPORTS_THUMBNAIL`. Документ з тим самим ідентифікатором `COLUMN_DOCUMENT_ID` може знаходитися в декількох каталогах.

4.6.2. Потік управління

Як було сказано вище, модель даних постачальника документів заснована 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. 4.7 наведено приклад того, як додаток обробки фотографій може використовувати SAF для доступу до збережених даних.

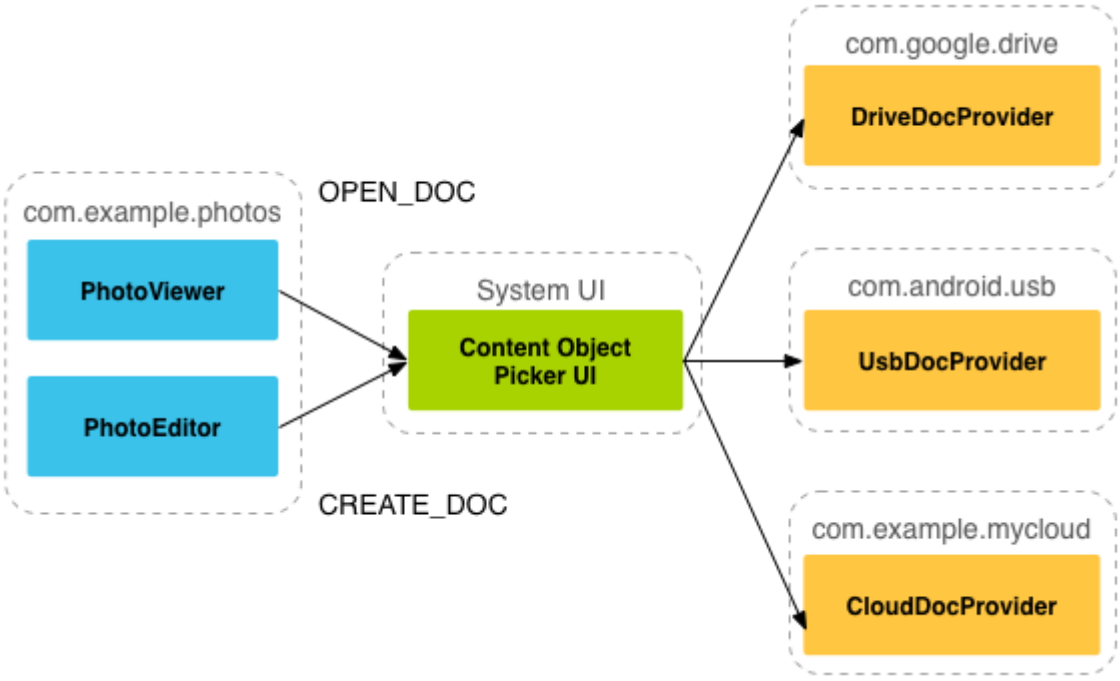

Рисунок 4.7 – Потік управління Storage Access Framework

Зверніть увагу на таке:

• На платформі SAF постачальники і клієнти не взаємодіють безпосередньо. Клієнт запитує дозвіл на взаємодію з файлами (тобто на читання, редагування, створення або видалення файлів).

• Взаємодія починається, коли додаток (в нашому прикладі обробляє фотографії) активізує намір `ACTION_OPEN_DOCUMENT` або `ACTION_CREATE_DOCUMENT`. Намір може включати в себе фільтри для уточнення критеріїв, наприклад, «надати відкриватися файлам з MIME-типом image».

- Коли намір спрацьовує, системний елемент вибору переходить до кожного зареєстрованого постачальника і показує користувачеві кореневі каталоги з контентом, відповідним до запиту.
- Елемент вибору надає користувачеві стандартний інтерфейс, навіть якщо постачальники документів значно різняться. Як приклад на рис. 4.7 зображені Диск Google, постачальник на USB-накопичувачі і хмарний постачальник.

На рис. 4.8 наведено елемент вибору, в якому користувач для пошуку зображень вибрав обліковий запис Диск Google.

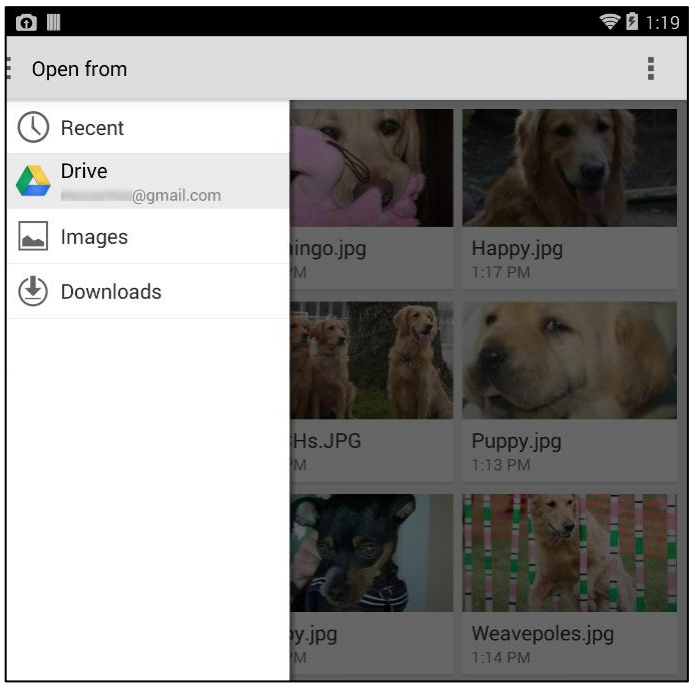

Рисунок 4.8 – Елемент вибору

Коли користувач вибирає Диск Google, зображення відображаються так, як показано на рис. 4.9. З цього моменту користувач може взаємодіяти з ними будь-якими способами, які підтримуються постачальником і клієнтським додатком.

4.6.3 Створення клієнтської програми

В Android версії 4.3 і нижче для того, щоб додаток міг отримувати файл від іншого додатка, він повинен активізувати намір, наприклад, `ACTION_PICK` або `ACTION_GET_CONTENT`. Після цього користувач повинен має вибрати якийсь один додаток, щоб отримати файл, а той, в свою чергу, повинен надати користувачеві інтерфейс, за допомогою якого він зможе вибирати і отримувати файли.

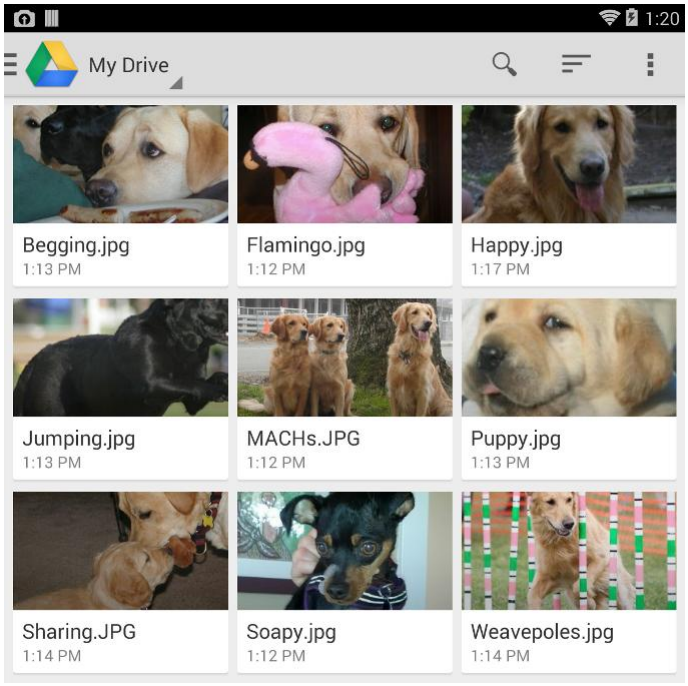

Рисунок 4.9 – Зображення

Починаючи з Android 4.4 і вище, у розробника є додаткова можливість – намір `ACTION_OPEN_DOCUMENT`, який відображає користувацький інтерфейс елемента вибору, керованого системою. Цей елемент надає користувачеві можливіть огляду всіх файлів, доступних в інших додатках. Завдяки цьому єдиному інтерфейсу, користувач може вибрати файл в 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:

1. Використовувати `ACTION_GET_CONTENT`, коли програму потрібно просто прочитати або імпортувати дані. При такому підході додаток імпортує копію даних, наприклад файл із зображенням.

2. Використовувати `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 постачальнику документів.

У цьому розділі показано, як написати клієнтську програму, що використовує наміри `ACTION_OPEN_DOCUMENT` і `ACTION_CREATE_DOCUMENT`.

Пошук документів. У наступному фрагменті коду намір `ACTION_OPEN_DOCUMENT` використовується для пошуку постачальників документів, що містять файли зображень:

```
private static final int READ_REQUEST_CODE = 42; ...
/ ** * Запускає намір розкрутити «вибір файлу» UI і вибрати зобра-
ження. * /
public void performFileSearch () {

    // ACTION_OPEN_DOCUMENT намір вибрати файл через файл системи //
браузер
    Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);

// Фільтр,що 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 зображень, використовуючи тип
MIME даних зображення. // Якщо хтось хоче знайти Ogg Vorbis файлів,
тип буде «аудіо / OGG». // Для пошуку всіх документів, доступних че-
рез встановлений провайдер зберігання,
 // це було б "* / *".
intent.setType ("image/*");

startActivityForResult (intent, READ_REQUEST_CODE);}
```

Зверніть увагу на таке.

- Коли додаток активізує намір `ACTION_OPEN_DOCUMENT`, він запускає елемент вибору, який відображає всіх постачальників документів, що відповідають заданим критеріям.
- Додавання категорії `CATEGORY_OPENABLE` у фільтри намірів призводить 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.

Обробка результатів. Коли користувач вибирає документ в елементі вибору, викликається метод `onActivityResult()`. Ідентифікатор URI, який вказує на обраний документ, міститься в параметрі `resultData`. Щоб витягти URI, слід викликати `getData()`. Цей URI можна використовувати для отримання документа, потрібного користувачеві. Наприклад:

```
@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 == Activi-
ty.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);
        }
    }
}
```

Вивчення метаданих документа. Маючи в своєму розпорядженні URI документа, розробник отримує доступ до його метаданих. У наступному фрагменті коду метадані документа, що визначається ідентифікатором URI, зчитуються і записуються в журнал:

```
public void dumpImageMetaData(Uri uri) {

    // Запит, так як це відноситься тільки до одного документу, буде
тільки повернення
    // один рядок. Там немає необхідності фільтрувати, сортувати,
або вибрати поля, так як ми хочемо
    // всі поля для одного документа.
    Cursor cursor = getActivity().getContentResolver()
            .query(uri, null, null, null, null, null);

    try {
    // moveToFirst () повертає брехня, якщо курсор має 0 рядків. Ду-
же зручно для
    // «якщо є щось дивитися, дивитися на нього» умовні.
        if (cursor ! = null && cursor.moveToFirst()) {

    // Зверніть увагу, що називається «Display Name». Це
    // постачальник-специфічним і може бути не обов'язково ім'я фай-
лу.
            String displayName = cursor.getString(
cursor.getColumnIndex(OpenableColumns.DISPLAY_NAME));
            Log.i(TAG, "Display Name:" + displayName);
```

```
            int sizeIndex = cur-
sor.getColumnIndex(OpenableColumns.SIZE);
            // Якщо розмір невідомий, значення, що зберігається
дорівнює нулю. Але так як
             // INT не може бути порожнім в Java, поведінка залежить
від конкретної реалізації,
             // який просто фантазія термін «непередбачуваний». з
тим
             // правило, перевірити, якщо це нуль перед призначенням
на міжнар. Це буде
             // часто трапляється: API-інтерфейс для зберігання
дозволяє видалені файли, чиї
             // розмір не може бути локально відомо.
            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();
    }
}
```

Відкриття документа. Отримавши URI документа, розробник може відкривати його і в цілому робити з ним все, що завгодно.

Об'єкт растрових зображень. Наведемо 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;
}
```

Зверніть увагу, що не слід проводити цю операцію в потоці призначеного для користувача інтерфейсу. Її потрібно виконувати у фоновому режимі, за допомогою `AsyncTask`. Коли файл з растровим зображенням відкриється, його можна відобразити в віджеті `ImageView`.

Отримання об'єкта InputStream. Нижче наведено приклад того, як можна отримати об'єкт `InputStream` за ідентифікатором URI. У цьому фрагменті коду зчитуються в об'єкт рядкового типу:

```
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.();
}
```

Створення нового документа. Додаток може створити новий документ в постачальнику документів, використовуючи намір `ACTION_CREATE_DOCUMENT`. Щоб створити файл, потрібно вказати в намірі MIME-тип і ім'я файлу, а потім запустити його з унікальним кодом запиту. Про інше подбає платформа:

```
// Ось деякі приклади того, як ви могли б назвати цей метод.
// Перший параметр є тип MIME, а другий параметр є ім'ям
// з створюваного файлу:
//
// 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 тільки результати показують, що можуть бути
«відкритими», наприклад,
    // файл (на відміну від списку контактів або часових поясів).
intent.addCategory(Intent.CATEGORY_OPENABLE);

    //Створення файлу з запитуваною типом MIME.
intent.setType(mimeType);
intent.putExtra(Intent.EXTRA_TITLE, fileName);
startActivityForResult(intent, WRITE_REQUEST_CODE);
}
```

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 прапор SUPPORTS_DELETE, то документ можна видалити, наприклад:

```
DocumentsContract.deleteDocument (getContentResolver (), uri);
```

Редагування документа. Платформа SAF дозволяє редагувати текстові документи на місці. У наступному фрагменті коду активізується намір `ACTION_OPEN_DOCUMENT`, а категорія `CATEGORY_OPENABLE` використовується, щоб відображалися тільки документи, які можна відкрити. Потім проводиться подальша фільтрація, щоб відображалися тільки текстові файли:

```
private static final int EDIT_REQUEST_CODE = 44;
/** * Відкрити файл для запису і 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 тільки результати показують, що можуть бути «відкриті»,
такі як
    // файл (на відміну від списку контактів або часових поясів).
intent.addCategory(Intent.CATEGORY_OPENABLE);

    // Фільтр для відображення тільки текстові файли.
intent.setType("Text / plain");

startActivityForResult(intent, EDIT_REQUEST_CODE);
}
```

Далі, з методу `onActivityResult()` (див. «`Обробка результатів`») можна викликати код для виконання редагування. У наступному фрагменті коду об'єкт `FileOutputStream` отриманий за допомогою об'єкта класу `ContentResolver`. За замовчуванням використовується режим запису. Рекомендується запитувати мінімально необхідні права доступу, тому не треба запитувати читання/запис, коли програмі необхідно тільки записати у файл:

```
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();
    }
}
```

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 в 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 додатком. Додаток фактично бере постійний URI-дозвіл, пропонований системою. В результаті користувач отримує безперервний доступ до файлів з програми, незалежно від перезавантаження пристрою:

```
final int takeFlags = intent.getFlags()
            & (Intent.FLAG_GRANT_READ_URI_PERMISSION
            | Intent.FLAG_GRANT_WRITE_URI_PERMISSION);
// Перевіряє найсвіжіші дані.
getContentResolver().takePersistableUriPermission(uri, takeFlags);
```

Залишається один заключний крок. Можна зберегти останні URI-ідентифікатори, з якими працював додаток. Однак не виключено, що вони втратять актуальність, оскільки інший додаток може видалити або модифікувати документ. Тому слід завжди викликати `getContentResolver().takePersistableUriPermission()`, щоб отримувати актуальні дані.

4.6.4. Створення власного постачальника документів

При розробці 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 розділі показано, як це зробити.

Маніфест. Щоб реалізувати власний постачальник документів, необхідно додати в маніфест додатка таку інформацію:

1. Цільовий API-інтерфейс рівня 19 або вище.

2. Елемент `<Provider>`, у якому оголошується нестандартний постачальник сховища.

3. Ім'я постачальника, тобто, ім'я його класу з ім'ям пакета, наприклад: `com.example.android.storageprovider.MyCloudProvider`.

4. Ім'я центру постачальника, тобто ім'я пакета (в цьому прикладі – `com.example.android.storageprovider`) з типом постачальника контенту (`documents`). наприклад, `com.example.android.storageprovider.documents`.

5. Атрибут `android:exported`, встановлений в значення `"True"`. Необхідно експортувати постачальник, щоб його було видно іншим додаткам.

6. Атрибут `android:grantUriPermissions`, встановлений в значення `"True"`. Цей параметр дозволяє системі надавати іншим програмам доступ до контента постачальника.

7. Дозвіл `MANAGE_DOCUMENTS`. За замовчуванням постачальник доступний всім. Додавання цього дозволу в маніфест робить постачальник доступним тільки системі. Це важливо для забезпечення безпеки.

8. Атрибут `android:enabled`, що має логічне значення, визначене в файлі ресурсів. Цей атрибут призначений для відключення постачальника на пристроях під управлінням Android версії 4.3 і нижче. Наприклад: `android:enabled="@bool/atLeastKitKat"`. Крім включення цього атрибута в маніфест, необхідно зробити таке:

1) у файл ресурсів `bool.xml`, що розташований в каталозі `res/values/`, додати рядок

```
<bool name="AtLeastKitKat">false</ Bool>
```

2) у файл ресурсів `bool.xml`, що розташований в каталозі `res/values-v19/`, додати рядок

```
<bool name="AtLeastKitKat">true</ Bool>
```

9. Фільтр наміру `android.content.action.DOCUMENTS_PROVIDER`, дозволяє відобразити постачальників в елементі вибору, коли система буде шукати постачальників.

Нижче наведено уривки зі зразка маніфесту, що включає в себе постачальник:

```
<manifest... >
...<Uses-sdk
        android:minSdkVersion="19"
        android:targetSdkVersion="19" />
....<provider
      an-
droid:name="com.example.android.storageprovider.MyCloudProvider"
      an-
droid: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>
```

Підтримка пристроїв під управлінням Android версії 4.3 і нижче. Намір `ACTION_OPEN_DOCUMENT` доступний тільки на пристроях з Android версії 4.4 і вище. Якщо додаток має підтримувати `ACTION_GET_CONTENT`, щоб обслуговувати пристрої, які працюють під управлінням Android 4.3 і нижче, необхідно відключити фільтр наміру `ACTION_GET_CONTENT` у маніфесті для пристроїв з Android версії 4.4 і вище. Постачальник документів і намір `ACTION_GET_CONTENT` слід вважати 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 наміру `ACTION_GET_CONTENT` на пристроях з Android версії 4.4 і вище рекомендується таким чином:

1. У файл ресурсів `bool.xml`, розташований в каталозі `res/values/`, додати наступний рядок:

```
<bool name="AtMostJellyBeanMR2">true</ Bool>
```

2. У файл ресурсів `bool.xml`, розташований в каталозі `res/values-v19/`, додати наступний рядок:

```
<bool name="AtMostJellyBeanMR2">false</ Bool>
```

3. Додати псевдонім операції, щоб відключити фільтр наміри ACTION_GET_CONTENT для версій 4.4 (API рівня 19) і вище. Наприклад:

```
<! – Цей псевдонім активності додається таким чином, щоб GET_CONTENT намі-
ри фільтр може 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>
```

Контракти. Як правило, при створенні нестандартного постачальника контенту одним із завдань є реалізація класів-контрактів. Клас-контракт являє собою клас `public final`, в якому містяться визначення констант для URI, імен стовпців, типів MIME та інших метаданих постачальника. Платформа SAF надає розробнику наступні класи-контракти, а тому 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:

```
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, Docu-
ment.COLUMN_MIME_TYPE,
        Document.COLUMN_DISPLAY_NAME, Document.COLUMN_LAST_MODIFIED,
        Document.COLUMN_FLAGS, Document.COLUMN_SIZE,};
```

Створення підкласу класу DocumentsProvider. Наступним кроком в розробці власного постачальника документів є створення підкласу абстрактного класу `DocumentsProvider`. Як мінімум, необхідно реалізувати такі методи:

- `queryRoots();`

- queryChildDocuments();
- queryDocument();
- openDocument().

Це єдині методи, реалізація яких суворо обов'язкова, проте існує набагато більше методів, які, можливо, теж доведеться реалізувати. Подробиці наводяться в описі класу DocumentsProvider.

Реалізація методу queryRoots. Реалізація методу queryRoots() повинна повертати об'єкт Cursor, який вказує на всі кореневі каталоги постачальників документів, використовуючи стовпці, визначені в DocumentsContract.Root.

У наступному фрагменті коду параметр projection представляє конкретні поля, потрібні об'єкту, що викликається. У цьому коді створюється курсор, і до нього додається один рядок, що відповідає одному кореневому 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.

```
@Override
public Cursor queryRoots(String[] projection) throws
FileNotFoundException {

    // Створення курсора або з запитуваних полів або за замовчуван-
ням
    // проекція, якщо «проекція» дорівнює нулю.
    final MatrixCursor result =
            new MatrixCursor(resolveRootProjection(projection));

    // Якщо користувач не увійшов в систему, повертає порожній коре-
невої курсор. Це усуває OUR
    // постачальник зі списку цілком.
    if (!isUserLoggedIn()) {
        return result;
    }

    // Можна мати кілька коренів (наприклад, для декількох облікових
записів в
    // те ж саме додаток) - просто додайте кілька рядків курсора.
    // 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 ко-
ренем опор
   // створення документів. FLAG_SUPPORTS_RECENTS означає, щонай-
```

```
менш, один каталог під коренем підтримує
    // Створення документа.
    // FLAG_SUPPORTS_SEARCH дозволяє користувачам шукати всі докумен-
ти додатки
    // розділяють
row.add(Root.COLUMN_FLAGS, Root.FLAG_SUPPORTS_CREATE |
            Root.FLAG_SUPPORTS_RECENTS |
            Root.FLAG_SUPPORTS_SEARCH);

    // COLUMN_TITLE це корнева назва (наприклад, галерея, Drive).
row.add(Root.COLUMN_TITLE, getContext().getString(R.string.title));

    // Цей ідентифікатор документа не може змінитися, як тільки він
розподілений.
row.add(Root.COLUMN_DOCUMENT_ID, getDocIdForFile(mBaseDir));

    // Дочірні типи MIME використовуються для фільтрації коріння і
присутній тільки в
    // Корені користувачів, які містять потрібний тип десь в їх ієр-
архії 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;
}
```

Реалізація методу queryChildDocuments. Реалізація методу `queryChildDocuments()` повинна повертати об'єкт `Cursor`, який 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 додатка. Метод отримує документи-нащадки каталогу на рівн, що нижче від кореневого. Його можна викликати на будь-якому рівні файлової ієрархії, а не тільки в кореневому каталозі. У наступному фрагменті коду створюється курсор з запитаними стовпцями. Потім в нього заноситься інформація про кожного найближчого нащадка батьківського каталогу. Нащадком може бути зображення, ще один каталог, загалом, будь-який файл:

```
@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 ім'я файлу відображення, MIME тип, розмір і так
далі.
        includeFile(result, null, file);
    }
    return result;
}
```

Реалізація методу queryDocument. Реалізація методу `queryDocument()` повинна повертати об'єкт `Cursor`, який вказує на заданий файл, використовуючи стовпці, визначені в `DocumentsContract.Document`.

Метод `queryDocument()` повертає ту ж інформацію, яку повертав `queryChildDocuments()`, але для конкретного файлу:

```
@Override
public Cursor queryDocument(String documentId, String[] projection)
throws
        FileNotFoundException {

    // Створює курсор із запитуваною проекцією або проекцією за замов-
чуванням.
    final MatrixCursor result = new
            MatrixCursor(resolveDocumentProjection(projection));
includeFile(result, documentId, null);
    return result;
}
```

Реалізація методу openDocument. Необхідно реалізувати метод `openDocument()`, який повертає об'єкт `ParcelFileDescriptor`, що подає зазначений файл. Інші додатки зможуть скористатися поверненим об'єктом `ParcelFileDescriptor` для організації потоку даних. Система викликає цей метод, коли користувач вибирає файл, і клієнтський додаток запитує доступ нього, викликаючи метод `openFileDescriptor()`. Наприклад:

```
@Override
public ParcelFileDescriptor openDocument(final String documentId,
final String mode,
CancellationSignal signal) throws
        FileNotFoundException {
    Log.v(TAG, "OpenDocument, mode:" + mode);
    // Це нормально робити мережеві операції в цьому методі, щоб за-
вантажити документ,
    // до тих пір, як ви періодично перевіряти CancellationSignal.
```

```
Якщо у вас є
    // дуже великий файл для передачі по мережі, краще рішення може
    // 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 в
режимі запису.
        try {
            Handler handler = new
Handler(getContext().getMainLooper());
            return ParcelFileDescriptor.open(file, accessMode, han-
dler,
                        new ParcelFileDescriptor.OnCloseListener() {
                @Override
                public void onClose(IOException e) {

                    // Прикріплення близького слухача, якщо документ
відкритий в режимі запису.
                    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);
    }
}
```

Безпека. Припустимо, що постачальник документів являє собою захищену паролем службу зберігання в хмарі, а додаток має переконатися, що користувач увійшов 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. Іншими словами, це повинен бути порожній кореневий курсор:

```
public Cursor queryRoots(String[] projection) throws
FileNotFoundException {
...
// Якщо користувач не увійшов в систему, повертає порожній кореневої
курсор. Це усуває OUR
```

```
     // постачальник зі списку цілком.
    if (!isUserLoggedIn()) {
        return result;
}
```

Наступний крок полягає у виклику методу `getContentResolver().notifyChange()`.

Пам'ятаєте об'єкт `DocumentsContract`? Скористаємося ним для створення відповідного URI. У наступному фрагменті коду система сповіщається 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 вище. Це гарантує, що документи постачальника будуть доступні тільки користувачам, що ввійшли в постачальник:

```
private void onLoginButtonClick() {
loginOrLogout();
getContentResolver().notifyChange(DocumentsContract
            .buildRootsUri(AUTHORITY), null);
}
```

Запитання для самоконтролю

1. Назвіть основні типи сховищ для даних, що доступні мобільному додатку на платформі Android.
2. Яким є призначення постачальника контактів?
3. Що таке локальна база даних SQLite?
4. Опишіть процес отримання даних за допомогою завантажувачів (Loaders).
5. Опишіть можливості при доступі до вбудованих баз даних на прикладі «Постачальника контактів».
6. Опишіть можливості платформи доступу до сховищ SAF.

СПИСОК ЛІТЕРАТУРИ

1. Google Play Hits 1 Million Apps // http://mashable.com/2013/07/24/google-play-1-million/ 16.06.2015.

2. Android App Stats // http://www.androlib.com/appstats.aspx 16.06.2015.

3. Google: 3 Billion Android Apps Installed; Downloads Up 50 Percent From Last Quarter // http://techcrunch.com/2011/04/14/google-3-billion-android-apps-installed-up-50-percent-from-last-quarter/ 16.06.2015.

4. Smartphone OS Market Share, Q1 2015 // http://www.idc.com/prodserv/smartphone-os-market-share.jsp 16.06.2015.

5. Tools Overview // http://developer.android.com/tools/help/index.html 16.06.2015.

6. AIDE- IDE for Android Java C++ // https://play.google.com/store/apps/details?id=com.aide.ui 16.06.2015.

7. Java Editor // https://play.google.com/store/apps/details?id=air.JavaEditor 16.06.2015.

8. JavaIDEdroid // https://play.google.com/store/apps/details?id=ch.tanapro.JavaIDEdroid 16/06/2015.

9. The Professional Android IDE // http://www.jetbrains.com/idea/features/android.html 16.06.2015

10 NBAndroid // http://plugins.netbeans.org/plugin/19545/nbandroid 16.06.2015

11. Android Studio // http://developer.android.com/sdk/index.html 16.06.2015

12. Backup & restore Android apps using adb // http://jonwestfall.com/2009/08/backup-restore-android-apps-using-adb/

13. SDK Tools // http://developer.android.com/tools/sdk/tools-notes.html 16.06. 2015.

14. Dalvik Executable format // https://source.android.com/devices/tech/dalvik/dex-format.html 16.06.2015.

15. Android – Invoke JNI Based Methods (Bridging C/C++ And Java) // https://davanum.wordpress.com/2007/12/09/android-invoke-jni-based-methods-bridging-cc-and-java/ 16.06.2015

16. Native C applications for Android // http://benno.id.au/blog/2007/11/13/android-native-apps 16.06.2015.

17. Android NDK // https://developer.android.com/tools/sdk/ndk/index.html 16.06.2015.

18. SKIA graphics library in chrome: first impressions // http://www.atoker.com/blog/2008/09/06/skia-graphics-library-in-chrome-first-

impressions/ 16.06.2015.

19. João Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente Documenting APIs with Examples: Lessons Learned with the APIMiner Platform // http://homepages.dcc.ufmg.br/~mtov/pub/2013_wcre_apiminer.pdf

20. AndroWish //http://www.androwish.org/index.html/home 18.06.15.

21. App Inventor Book, Classic version // http://www.appinventor.org/book

22. RAD Studio XE8 //https://www.embarcadero.com/products/rad-studio

23. What you should know about Hypernext Android Creator (HAC)? // https://sites.google.com/site/androidappdevsindia/what-you-should-know-about-hypernext-android-creator-hac

24. Programming Made Simple // https://code.google.com/p/simple/ 18.06.2015.

25. How To Write A Simple Application // https://code.google.com/p/simple/wiki/HowToWriteASimpleApplication 18.06.15

26. Visual C++ Cross-Platform Mobile // https://www.visualstudio.com/en-us/features/cplusplus-mdd-vs.aspx 18.06.2015

27. Top 10 Android Apps and IDE for Java Coders and Programmers // https://blog.idrsolutions.com/2014/12/android-apps-ide-for-java-coder-programmers/ 18.06.2015

28. List of IDEs for Android App Development, Which is Best for You? // http://tekeye.biz/2014/list-of-android-app-development-ides 18.06.2015

29. Apps for Programming on Android // http://android.appstorm.net/roundups/developer/15-apps-for-programming-on-android/ 18.06.2015

30. The Perfect Platform for Game Developers: Android // http://www.developer.com/ws/android/client/the-perfect-platform-for-game-developers-android.html 18.06.2015

31. 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.

32. 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.

33. 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)

34. Бордюг В. Л. Выбор инструментов для разработки мобильного приложения методом анализа иерархии Т. Саати / В. Л. Бордюг, Е. Г. Панченко, О. Н. Трифонова // Nauka-rastudent.ru. – 2015. – No. 02 (014-2015) / [Электронный ресурс] – Режим доступа. URL: http://nauka-rastudent.ru/14/2419/

35. Саати Т. Принятие решений. Метод анализа иерархий / Т. Саати. – М. : Радио и связь. – 1993. – 278 с.

36. Introduction to Android // https://developer.android.com/guide/index.html 21.10.2016

37. User Interface // https://developer.android.com/guide/topics/ui/index.html 21.10.2016

38. App Components // https://developer.android.com/guide/components/index.html 21.10.2016

39. Голощапов А. Google android. Системные компоненты и сетевые коммуникации / А. Голощапов. – СПб. : БХВ-Петербург, 2012. – 384 с.

40. Голощапов А. Google android. Создание приложений для смартфонов и планшетных ПК / А. Голощапов. – СПб. : БХВ-Петербург, 2013. – 832 с.

41. Дэвид Гриффитс Head first. Программирование для android / Дэвид Гриффитс, Дон Гриффитс. – СПб. : Питер, 2016. – 704 с.

42. С. Хашими / Разработка приложений для Android / С. Хашими, С. Коматинени, Д. Маклин – СПб. : Питер, 2011. – 736 с.

Навчальне видання

ШМАТКО Олександр Віталійович
ПОЛЯКОВ Андрій Олександрович
ФЕДОРЧЕНКО Володимир Миколайович

АНАЛІЗ МЕТОДІВ І ТЕХНОЛОГІЙ РОЗРОБКИ МОБІЛЬНИХ ДОДАТКІВ ДЛЯ ПЛАТФОРМИ ANDROID.
ЧАСТИНА 2

Навчальний посібник

Відповідальний за випуск проф. Годлевський М. Д.
Роботу до видання рекомендував проф. Горілий О. В.

Редактор Л. Л. Яковлева

План 2017 р., поз.143

Підписано до друку р. Формат 60х84 1/16. Папір офсет.
Друк – ризографія. Гарнітура Times New Roman. Ум. друк. арк..
Наклад 50 прим. Зам № _____. Ціна договірна

Видавничий центр НТУ «ХПІ»
Свідоцтво про державну реєстрацію ДК № 3657 від 24.12.2009 р.
61001, Харків, вул. Кирпичова, 2
