

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 закриття служби 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 іншими компонентами 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 елемент `<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 включити 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 дозволяє 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. Снадкування 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 таке:

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. Снадкування 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 потік, 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 користувацького інтерфейсу.
        HandlerThread thread = new
        HandlerThread("ServiceStartArguments",
            Process.THREAD_PRIORITY_BACKGROUND);
        thread.start();
        // Отримуємо HandlerThread і використовуємо 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 запуску запиту, відправляємо повідомлення, щоб
        // почати
    }
}

```

```

// роботу і відправляємо ID, так що ми знаємо, який запит ми зупинили, 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 повернуємо 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, який 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 створить службу і викличе `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 відновлена 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 (`HelloService`), 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 коли-небудь 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 фонові 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 видалити 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, фонові музичні служби можуть бути запущені 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 реалізовувати 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;          // визначаємо 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 повернен-

ня з методу `onStartCommand()`). Якщо служба є прив'язаною, активний життєвий цикл закінчується, коли повертається метод `onUnbind()`.

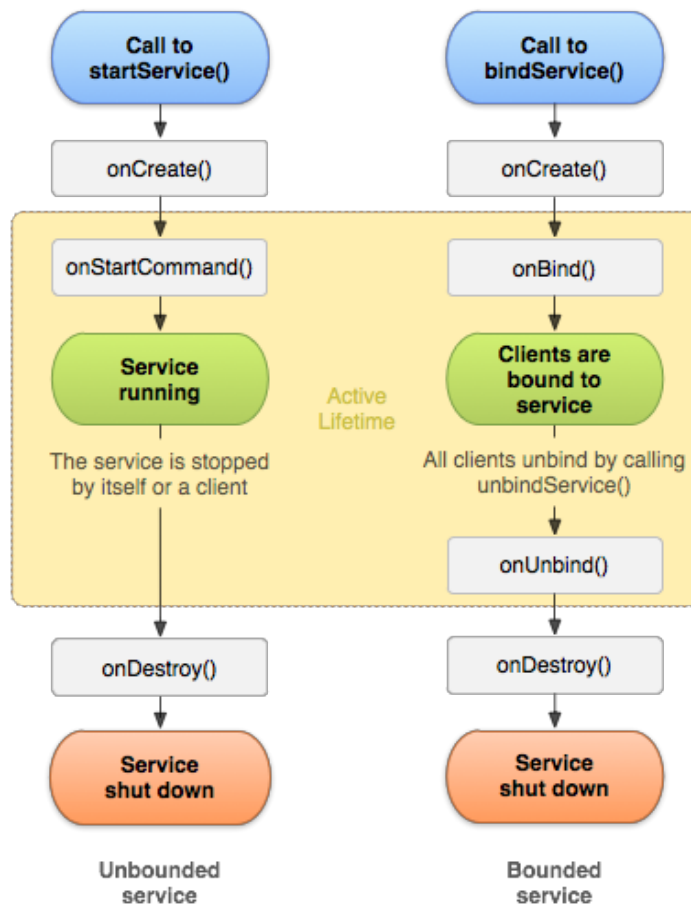


Рисунок 3.1 – Життєвий цикл служби

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 onClick(View v) {
        if (mBound) {
            // Виклик методу з LocalService.
            // Однак, якщо цей виклик був те, що може викликати зависан-
            // ня, то цей запит повинен
            // відбуватися в окремому потоці, щоб уникнути уповільнення
            // роботи дії.
            int num = mService.getRandomNumber();
            Toast.makeText(this, "number: " + num,
                Toast.LENGTH_SHORT).show();
        }
    }

    /** Визначає функцію зворотного виклику для

```

```

зв'язування служби, 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 об'єкт `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 використо-
            вувати
            // для взаємодії 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()`. Інтерфейс `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 відмінити 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 служби 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 явниму 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 службу, 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.

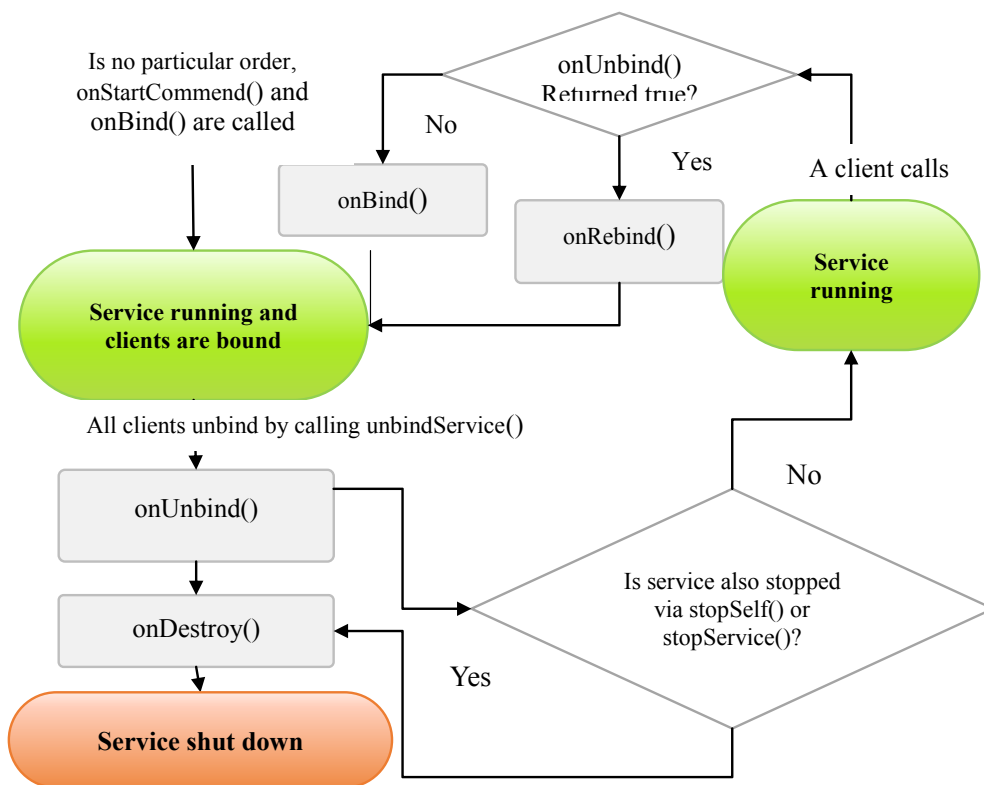


Рисунок 3.2 – Життєвий цикл запущеної служби, для якої виконується прив'язка

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 плані процесів?
5. Опишіть процес прив'язки до служби.