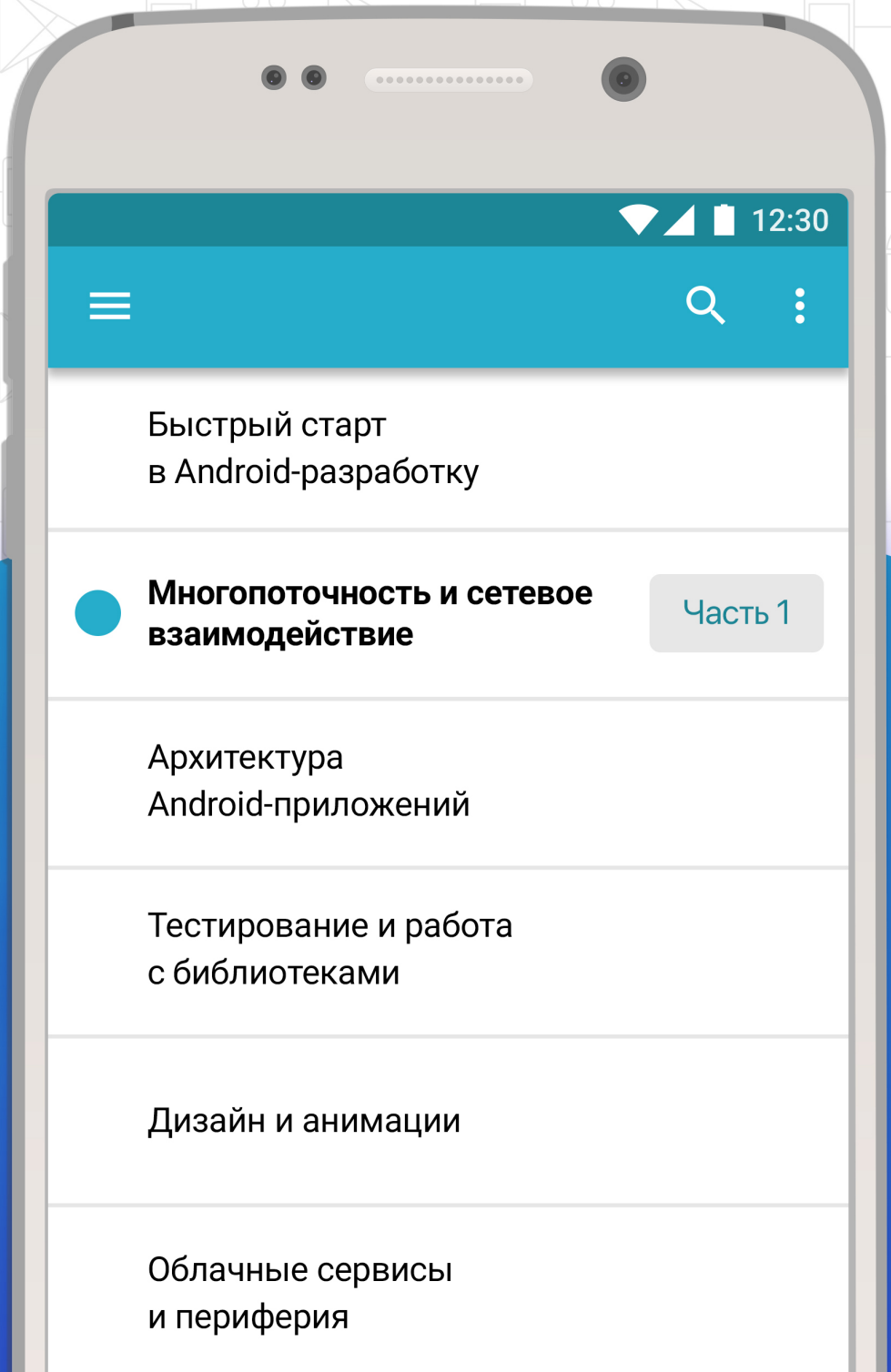
фонд развития
онлайн образования
eldf.ru

e·legion

academy.e-legion.com

# Программа Android-разработчик

## Конспект



# Оглавление

# Глава 1

# НЕДЕЛЯ 1

## 1.1. Обзор средств для обеспечения многопоточности

### 1.1.1. Знакомство с курсом

Привет и добро пожаловать во второй блок для разработки под Android.

Вы уже знаете и умеете работать с такими сущностями как:

- Что такое Context;
- Жизненный цикл Activity, запуск явных и неявных Intent;
- Как работать с Fragment и FragmentManager;
- Станддартные типы View и ViewGroup;
- Сохранение в SharedPreferenced;
- Система сборки Gradle.

Однако этого недостаточно, чтобы написать хоть какое-то приложение, которое будет востребованно и конкурентно способно. Это нужно исправить. Поэтому в этом блоке мы с вами изучим:

- Инструменты для обеспечения многопоточности;
- Работа со списками;

- Работа с файлами и БД;
- Сетевое взаимодействие;
- Реактивное программирование.

Кроме того, вас ожидает много практики. Не будем терять попусту время. Поехали!

### 1.1.2. Многопоточность и параллельное программирование

Поговорим о том, что такое многопоточность. И начнем с основ. Что такое процесс?

***Процесс*** – *глобальная сущность, выделенные ресурсы* – адресное пространство в памяти и время на выполнение. Другими словами, каждая программа запускается в отдельном процессе. Процесс не может просто так воспользоваться чужими ресурсами – например, памятью. Для этого нужно настроить *межпроцессное взаимодействие*, но это отдельная тема, ее в данном курсе рассматривать не будем.

Кроме того в процессе существуют потоки. ***Поток*** – *внутренняя сущность процесса, выполняет программный код* – сущность, которая выполняет программные инструкции.

Допустим, в нашем процессе мы создаем несколько потоков, которые занимаются своими задачами. Пока они работают с разными переменными, проблем нет: каждый живет в своем собственном мирке. Например, один поток рисует интерфейсы, а второй – выполняет сетевой запрос. Но если потоки начинают использовать одну переменную, то возникают проблемы.

Рассмотрим программу, которая считает людей в доме. Мы создаем по потоку для каждого этажа, создаем общую переменную, которую эти потоки будут инкрементировать.

```
1  private static int mCount = 0;
2
3      public static void main(String[] args) {
4          for (int x = 0; x < 10; x++) {
5              new Thread(new Runnable() {
6                  @Override
7                  public void run() {
8                      for (int y = 0; y < 100; y++) {
9                          mCount++;
```

```
10                          System.out.println(mCount + " " +
                             ↪  Thread.currentThread().getName());
11                          try {
12                              TimeUnit.MILLISECONDS.sleep(100);
13                          } catch (InterruptedException e) {
14                              e.printStackTrace();
15                          }
16                      }
17                  }
18              }).start();
19          }
20      }
```

Наверняка вы удивитесь результату – он будет меньше ожидаемого. Проблема в том, что если потоков больше, чем процессов, то процессор переключается между потоками для того, чтобы они выполнялись более или менее параллельно. Только момент переключения случаен, так же, как и время. В данном примере первый поток может считать значения из переменной, увеличивать ее, но не успеть его записать, потому что процессор переключился на другой поток. Второй поток считает то же самое значение, который считал и первый поток, увеличит это значение, но запишет. Затем процессор переключится обратно на первый поток, который продолжится на том месте, где остановился – перед записью увеличенного значения в переменную. То есть, другими словами, вычисления второго потока просто исчезнут. Для того, чтобы избежать таких проблем, существуют механизмы синхронизации.

**Синхронизация** – это механизм, который позволяет потоку спокойно завершить работу, без прерывания другими потоками. Достигается разными методами в зависимости от языка программирования, платформы и намерения программиста. Любой объект может быть **«mutex-ом»** – идентификатором того, что поток занят или свободен. Но это не единственные способы синхронизации в **Java** – мы их рассмотрим далее.

Кстати, если рассмотреть предыдущий пример, то мы могли бы выделить несколько потоков, которые записывали количество людей в свою локальную переменную, а затем синхронизировано увеличивали общий счетчик на рассчитанное количество. Вы можете попробовать реализовать такую программу, используя экзекьютеры. Разделение «один этаж – один поток», позволяет запустить несколько потоков параллельно. Ведь они не мешают друг другу увеличивать свои собственные счетчики. А изменение общего счетчика один раз за этаж вместо количества людей на этаже хорошо скажется на производительности.

А что с Android? С ним все «просто». Есть **main thread** – главный поток, который отвечает за отрисовку интерфейсов. Чем меньше нагрузка на main thread, тем отзывчевее и плавнее интерфейс. И тем больше шансов, что пользователь не влепит одну звездочку. Поэтому задачу производительности, не связанную с отрисовкой интерфейса, желательно опрокинуть в фоновый поток. Это относится к тяжелым задачам: например, сетевым запросам, запросам в базу данных, тяжелым манипуляциям с данными или изображениями. Разработчики самого Android стараются надавить на разработчиков приложений под Android, чтобы они не захламляли main thread. Так, в Android 3.0 Honeycomb уровень 11 появился network on main thread exception, который бросался при сетевом запросе из main thread, так как сетевой запрос – это тяжелая операция, она может занять несколько секунд, а если в Android приложение зависло на 5 секунд, то бросается **ANR ошибка** – приложение не отвечает. Поэтому было решено запретить сетевые запросы из main thread.

Параллельно с этим развитие платформы Android приводило к появлению новых инструментов многопоточного взаимодействия. Некоторые решения были спорными, некоторые хорошими, но вне зависимости от этого все их нужно знать. Правда, в современной разработке эти нативные решения постепенно уступают место сторонним библиотекам, которые реализуют тот или иной архитектурный паттерн.

Зачем мы уделяем время нативным решениям? Мы считает, что если разработчик в состоянии реализовать любую функциональность с помощью того, что предлагает Android Framework, то ему не составит труда разобраться в сторонней библиотеке. Плюс тащить библиотеку ради одного сетевого запроса вряд ли целесообразно, и, напротив, если проект ожидается большим или разрастается до большого, то уже тогда имеет смысл подключать архитектурную библиотеку. Меньше слов – больше дела! Давайте пройдемся по инструментам обеспечения многопоточности в Java.

### 1.1.3. Обзор инструментов для обеспечения многопоточности в Java (Thread, Runnable, Callable, Future, Executors)

В этой главе мы кратко пройдемся по инструментам класса, с помощью которых обеспечивается многопоточность в Java. Для создания нового потока нужно вызвать конструктор **New thread**. Чтобы запустить этот новый поток, нужно запустить на нем метод Start. Тогда выполнится код метода Run. Чтобы поток делал что-то полезное, нужно передать ему что-то в конструктор объект типа Runnable, вписать в Run, что нужно сделать или переопределить метод Run и что-то тоже туда вписать. Поток живет, пока выполняется в методе Run, после поток уничтожает-

ся. И обратите внимание на тип возвращаемого значения в методе Run – это void – ничего не возвращается.

```
public class MyThread extends Thread {
    @Override
    public void run() {
     //do something
    }
}

new Thread(new Runnable(){
    @Override
    public void run() {
     //do something
    }
});

new MyThread().start();
```

Теперь поговорим про синхронизацию потоков. Это нужно, чтобы несколько потоков могли безопасно оперировать общими переменными. Начнем со слова «synchronized» – это самый простой способ сделать метод или блок кода в методе потокобезопасным. Synchronized гарантирует, что метод или блок кода, помеченные этим словом, будут выполняться только в одном потоке одновременно. Причем можно выбирать, сделать код синхронизированным на уровне объекта...

```
public synchronized void doSomething(){
    //Метод синхронизован на уровне объекта
}
public void doSomethingBlock(){
    //...
    synchronized (this){
    //Блок синхронизован на уровне объекта
}
```

...или же на уровне класса:

```
private final static Object lock = new Object();
public void doSomethingStaticBlock(){
    //...
    synchronized (lock){
    //Блок синхронизован на уровне класса с помощью статического объекта lock
    }
    //...
}
```

Для блокировки на уровне класса нужно, чтобы метод был статическим либо передавать в synchronized сам класс или статический объект. Разница между синхронизированным методом и блоком в том, что блок требует объект в качестве монитора. И если у нас, например, два метода, которые должны быть синхронизированными, но при этом нет опасности в том, что каждый метод может вызываться отдельно в разных потоках одновременно, то можно тело каждого из методов обернуть в synchronizedBlock и передавать разные объекты в качестве мониторов.

В многопоточном программировании возможны ситуации, когда выполнение работы одного потока зависит от работы другого потока. Рассмотрим следующую ситуацию: поток 1 входит в синхронизированный метод и блокирует этот метод для других потоков и осознает, что ему не хватает каких-то условий для выполнения своего кода. Тогда вызывается метод Wait, он освобождает монитор и переводит себя в режим ожидания. Поток два заходит в синхронизированный метод и захватывает монитор. Если предположить что у него все хорошо с условиями, он хорошо выполняет свою работу и в конце вызывает Notify, тем самым уведомляя поток один, что ему можно работать дальше. Соответственно, поток 2 заканчивает выполнять свой код, покидает синхронизированный метод и освобождает монитор. И только после этого поток 1 продолжает свою работу со следующего после wait выражения. Вот и все.

Разница между Notify и NotifyAll: в случае Notify система сама случайно выбирает, какой поток из ожидающих запустить, а NotifyAll запускает их сразу все – потоки соревнуются, кто захватит монитор.

Кроме того, возможна ситуация, в которой несколько потоков будут зависеть друг от друга: каждому потоку нужно будет, чтобы другой поток выполнил свой код. Это называется **DeadLog** – это плохо и этого следует избегать.

Если переменная помечена как Volatile, то изменение в одном потоке сразу становится заметно в других потоках. То есть простыми словами, когда поток оперирует с внешней переменной, на самом деле он оперирует с ее копией, и в конце концов записывает значение копии в оригинал.

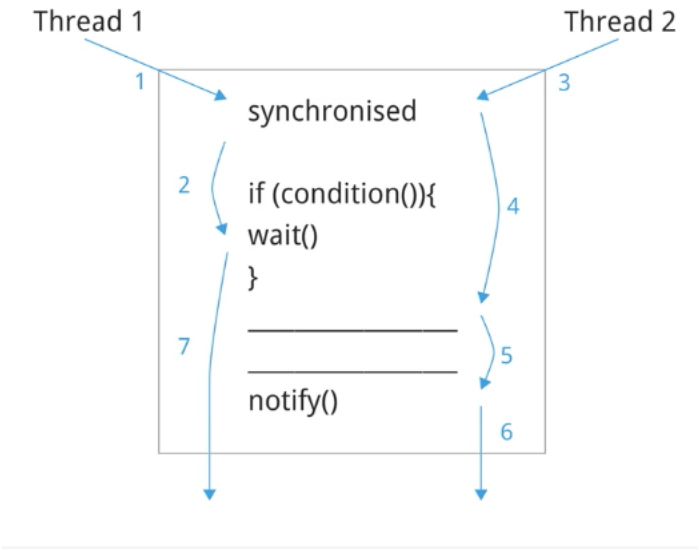



Рис. 1.1: Схема работы потоков

Причем может возникнуть такая ситуация, что поток будет прерван до того, как запишет новое значение копии в оригинал. Тогда с точки значение прервавшего потока в переменной будет старое значение, обновленное значение. Поток отработает с ним, после этого вернет управление первому потоку, а первый поток запишет значение из копии в оригинал. Volatile приводит к тому, что изменение переменной сразу будет видно в других потоках. Подробную информацию можно найти в документации Java.

Управлять потоками сложно. Сложно создавать потоки, проверять условия, синхронизировать, передавать задачи, получать результаты. К счастью, в Java есть Executor и ExecutorService. Executor добавляет новый уровень абстракции. Мы уже не создаем потоки с помощью NewThread, вместо этого мы пользуемся фабричными методами, получая сформированный pool потока. Далее мы передаем задачи Executor, и он уже распределяет их между потоками. Это очень удобный и популярный механизм многопоточного программирования.

В Submit можно передать объект типа Callable. Callable, в отличие от Runnable, возвращает результат типа future. Future – это что-то вроде контейнера для окончательного результата. И если метод из Done вернул True, то можно достать результат из future с помощью get. Можно просто вызвать метод get, но этот метод блокирующий – возврат значения займет столько времени, сколько потребуется.

```
ExecutorService executorService  =  Executors.newFixedThreadPool(4);

Future <String> future =  executorService.submmit(new Callable<String>){
    @Override
    public String call() throws Exception {
    //do something
    return result;
    }
});

while (!future.isDone()){
    TimeUnit.Milliseconds.sleep(100)
}
String result = future.get()
```

И последний в этом блоке – пакет java.util.concurrent. В concurrent хранится много классов, включая те, что рассмотрены выше, которые облегчают многопоточную работу. Тут есть и синхронизированные коллекции, и замки, и atomic объекты, и различные синхронизаторы. Куча объектов, которые заточены решать распространенные и не очень задачи. И если перед вами встанет задача сделать что-то многопоточное по-хитрому, не поленитесь, а пройдитесь сначала по пакету java.util.concurrent и, возможно, вы найдете быструю, простую и надежную реализацию решения вашей проблемы.

Далее представлена картинка блокирующей очереди.

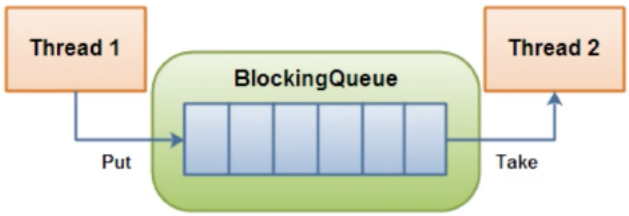



Рис. 1.2: Блокирующая очередь

Один поток кладет данные в очередь, а второй их считывает.

Где искать дополнительную информацию?

- Официальная документация Oracle
- Статьи на habrahabr.ru
- *«Java Concurrency in Practice»*
- Java 8 Reactive streams, ComletableFuture
- RxJava

Поиск по ключевым словам в Google выдаст вам много ссылок, личные сайты программистов, а так же видео на Youtube с выступлениями на конференциях. Источников очень много, нужно выбрать тот, который вам нравится.

Из книг стоит упомянуть *«Java Concurrency in Practice»*. В этой книге можно найти вообще все.

Также тут не было упомянуто о классах, которые появились Java 8 Reactive streams, ComletableFuture и т.д. Android начал частичную поддержку Java 8, но она меньше, чем хотелось бы. Для получения всех доступных пакетов и фишек нужно понимать минимальный уровень поддерживаемых IP. И если это оправданно в личных проектах, то большие компании такого себе позволить не могут. Но это не повод игнорировать Java 8.

И еще RxJava – реактивному программированию посвящена целая неделя курса.

### 1.1.4. Обзор инструментов для обеспечения многопоточности в Android (IntentService + BroadcastReceiver, HaMeR, AsyncTask, Loaders)

В этой главе мы изучим многопоточность средствами Android: способов много и каждый из них имеет свою цель и справляется с определенными типами задач лучше других. Рассмотрим краткий обзор всех инструментов.

1. Если пройтись по компонентам Android, то вроде как решение вопроса многопоточности напрашивается само собой – **Service**. Service можно настроить на запуск в отдельном потоке, и если чуть-чуть пошаманить, то он будет работать даже в свернутом приложении. А если еще больше пошаманить, то она будет работать между приложениями. Service чаще

всего работают в связке с **BroadcastReceiver**, чтобы можно было уведомлять activity о прогрессе задачи или о том, что работа завершена. Проблема состоит в отсутствии гибкости и сложности всего подхода. Service хорошо подходит, если нам нужно скачать файл или отследить местоположение, то есть для задач, в которых не требуется участие пользователя.

2. **AsyncTask** – инструмент, который был добавлен для решения недолгих задач. Его преимущество в простоте интерфейса, явное разделение на то, что должно быть выполнено до фоновой операции, в фоне и после фоновой операции, а также как отображать прогресс операции. Однако из-за особенностей реализации с AsyncTask легко словить утечку контекста или, в худшем случае, даже crash. Мы разберем этот раздел чуть позже.

3. **HandlerThread**, который работает вместе со связкой **Handler Message Runable** – **HaMeR**. HandlerThread используется для гибкого, за счет своей низкоуровневости, взаимодействия с главным потоком. Взаимодействие происходит за счет обмена сообщениями. Эта тема сложна для восприятия, но она очень важная. Мы также полностью разберем, как работает main thread.

4. Разберем также **Loader** – механизм столь же мощный, сколь и сложный. Они зависят от контекста приложения, что в частности означает то, что их можно запустить на одном экране, а результат получить на другом. Мы рассмотрим работу с ними на занятии, посвященному спискам.

И теперь, когда фронт работ намечен, давайте разбираться подробнее.

### 1.1.5. Материалы для самостоятельного изучения

1. https://youtu.be/zxZ0BXlTys0
2. https://goo.gl/h2SoAX
3. https://goo.gl/3LdUkM
4. https://goo.gl/bNsf6q
5. https://habrahabr.ru/company/luxoft/blog/157273/
6. http://www.baeldung.com/java-util-concurrent

7. https://docs.oracle.com/javase/tutorial/essential/concurrency/
8. https://www.ozon.ru/context/detail/id/3174887/

## 1.2. Service + BroadcastReceiver

### 1.2.1. Знакомство с Service, IntentService

В этом видео мы чуть более детально рассмотрим Service – компонент приложения, который используется для выполнения долгих операций, чаще всего в фоновом потоке, в бэкграунде, без взаимодействия с пользовательским интерфейсом. Также сервисы могут работать в другом процессе. В этом случае для связи с ними используется межпроцессное взаимодействие.

За примером работы сервиса далеко ходить не надо: достаточно включить музыку в любом приложении. Если музыка продолжает играть, даже если вы свернули приложение, значит, используется сервис.

```
<service
 android:name="CustomService"
 android:enabled="true"
 android:exposed="true"/>
```

В Android различают три вида сервисов:

- **Foreground-service** – сервис, работа которого важна для пользователя и внезапная остановка сервиса будет заметна и скорее всего воспринята негативно. К слову, проигрывание музыки относится к такому виду сервисов. Еще пример – скачивание файла. И вы, вероятнее всего, замечали, что и при скачивании файла, и при проигрывании музыки в статус-баре появляется иконка, связанная с нотификацией, которая дает знать пользователю, что в телефоне происходит какая-то деятельность. Наличие такой нотификации – уведомления – обязательное условие запуска foreground-service. Это требование системы. Foreground-service имеет один из самых высоких приоритетов среди процессов. Это означает, что система очень маловероятно прибьет его, если рабочему activity на экране не будет хватать ресурсов. Рабочий activity, к слову, имеет самый высокий приоритет.

- **Background-service** – сервис, работа которого не очевидна для пользователя. Это может быть сетевой запрос, запись в базу данных, сбор статистики и тому подобное. Запущенный сервис живет, пока не закончит свою работу, либо до вызова метода **stopService()**. Плюс мы не исключаем возможность быть прибитым системой. Background-service – наиболее распространенный вид сервиса. Для его запуска используется метод **startService()**, которому в качестве аргумента мы передаем **intent**, как и в случае с методом **startActivity()**.
- **Bound-service** – сервис, который с помощью интерфейса позволяет вызывающей стороне (activity, чаще всего) взаимодействовать с ним: отправлять запросы и получать результаты. Для привязки сервиса используется метод **bindService()**. Bound-service умрет только тогда, когда отвяжутся или будут убиты все компоненты, которые были к нему привязаны, если сервис не был запущен через метод **startService()**.

При создании сервиса чаще всего наследуются от одного из двух вариантов базового класса: это, собственно, сам Service – родоначальник всех сервисов и Intent-service – его упрощенная версия.

```
public class MyService extends Service{
    public CustomService() {
    }
    @Override
    @Nullable
    public IBinder onBind(Intent intent){
        return null;
    }
}
```

```
public class AnotherService extends IntentService{
    public AnotherService(String name) {
        super(name);
    }
    @Override
    protected void onHandleIntent(Intent intent){
        //do something with data
    }
}
```

В чем же разница? Дело в том, что по умолчанию сервис работает в главном потоке, однако

это не всегда желаемое поведение. При наследовании от Service создание потока, очистку после окончания работы и сами условия установки сервиса приходится обрабатывать самому. В качестве альтернативы используется intent-service – сервис, который сам создает рабочий поток, все входящие intent'ы обрабатывает в нем по очереди и завершает себя после обработки всех intent'ов. Работа с intent-service – это простой и достаточно эффективный способ организации фоновых операций.

Скажу пару слов о жизненном цикле сервиса. Он намного проще, чем жизненный цикл activity. Но может развиваться двумя разными путями, в зависимости от того, как он был запущен.

Разберем методы жизненного цикла поподробнее:

1. **onCreate().** Этот метод вызывается один раз при создании сервиса вне зависимости от того, как он был создан. В нем происходит первоначальная настройка сервиса, инициализация необходимых переменных, создание потоков и т.п.

2. Если сервис был запущен с помощью метода **bindService**, то после **onCreate()** вызовется метод **onBind()**. Этот метод возвращает объект интерфейса iBinder – как раз тот, с помощью которого будет происходить взаимодействие между сервисом и вызывающей его стороной.

3. Вызывающая сторона может отвязаться от сервиса через метод **unbindService()**. В этом случае у сервиса сработает callback **onUnbind()**, возвращающий булевые значения, который указывает, можно ли к сервису привязаться заново. При перепривязке сработает метод **onRebind()**

4. **onDestroy()** – метод, который вызывается при уничтожении сервиса. То есть если Intent-service закончил обрабатывать все intent'ы своей очереди, либо от сервиса отвязались все компоненты, которые были к нему привязаны, либо сервис был остановлен извне методом **stopService()** или самостоятельно методом **stopSelf()**, то вызовется метод **onDestroy()**. Это последний метод, который вызовется у сервиса. В нем нужно освободить ресурсы, например, остановить потоки, очистить переменные, отключить приемники и тому подобное, чтобы не оставлять мусора.

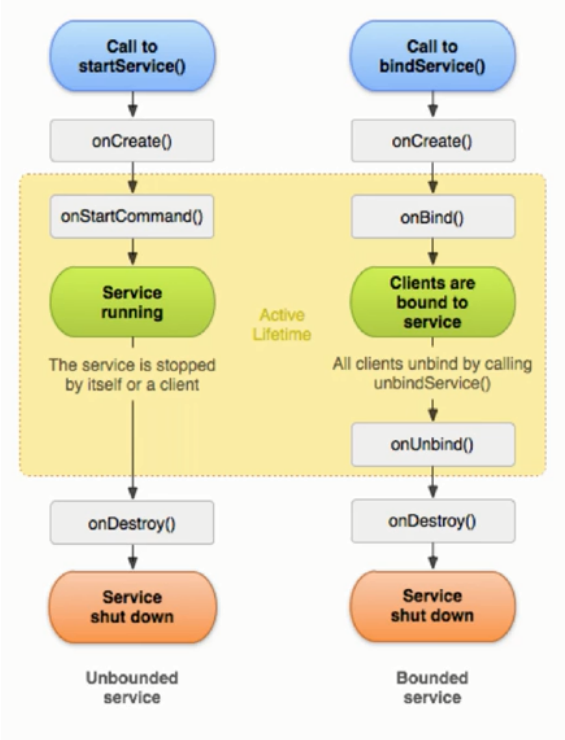



Рис. 1.3: Жизненный цикл сервиса

У сервиса можно выделить активное время жизни – время, когда он выполняет свою работу. У привязанного сервиса это период от **onBind()** до **onUnbind()**. У просто запущенного – от **onStart()** до **onDestroy()**.

### 1.2.2. Создание Service

Мы начинаем серию рассказов о service'ах, receiver'ах, уведомлениях и связке всего вышеперечисленного в одном приложении. Первым пунктом мы создадим сервис и добавим в него задачу. Откроем обычный пустой проект с empty activity. Далее мы заходим в **Java**, находим наш пакет, щелкаем правой, выбираем **New** –> **Service** –> **Service**.

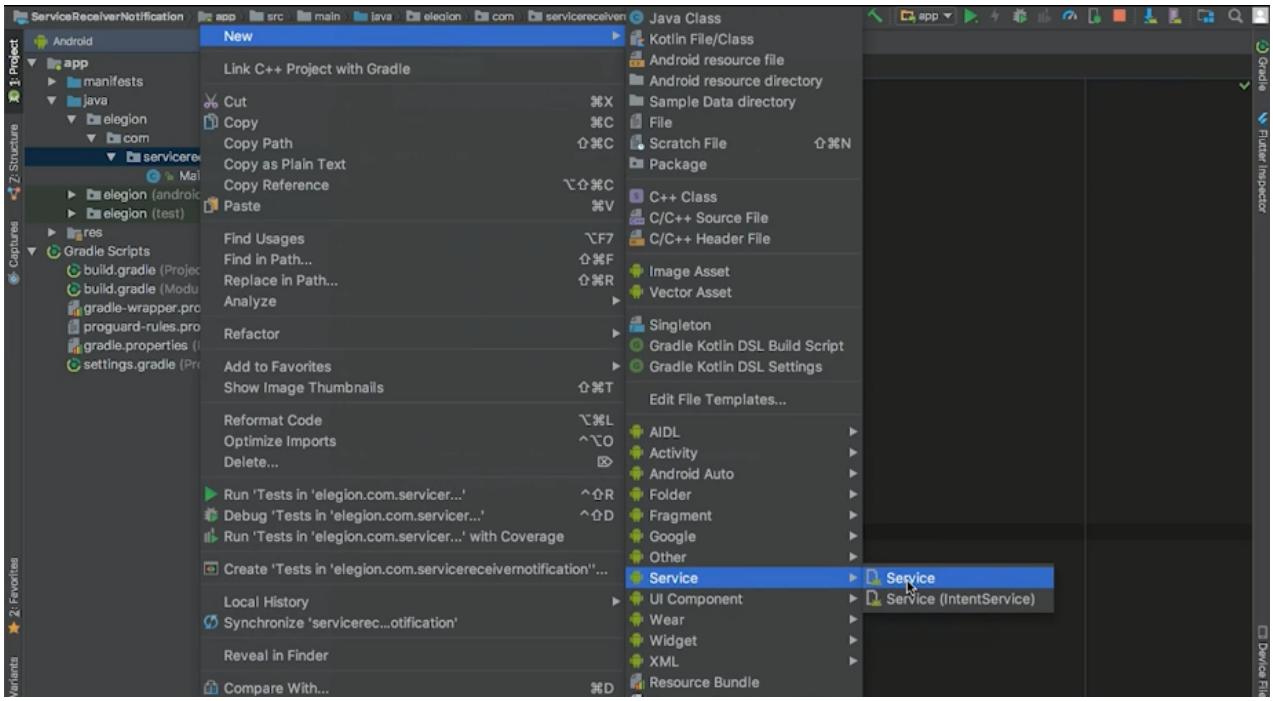



Рис. 1.4: Жизненный цикл сервиса

Выбираем ClassName. В нашем случае это будет CountService, он будет считать. Щелкаем **Finish**.

**onBind** возвращает **null**, так как **Bind** используется для привязанных сервисов, но их я оставлю вам на десерт. Дальше идет жизненный цикл: **onCreate, onStartCommand и onDestroy**. Пишем **Log.d**. Добавим TAG: psfs - public static final String TAG. Проверим, что в onCreate и onDestroy ничего не происходит и уберем super-методы. В **onStartCommand()** вернем START_STICKY. Он означает, что если сервис был убит системой, то она попытается его восстановить, как только у нее появятся для этого ресурсы.

```
1  import...
2      public class CountService extends Service {
3          public static final String TAG = CountService.class.getSimpleName();
4
5          public CountService() {
6          }
7      }
8
```

```
9       @Override
10      public IBinder onBind(Intent intent){
11          return null;
12      }
13
14      @Override
15      public void onCreate(){
16          Log.d(TAG, msg: "onCreate: ");
17      }
18
19      @Override
20      public int onStartCommand(Intent intent, int flags, int startId){
21           Log.d(TAG, msg: "onStartCommand: ");
22           return START_STUCKY;
23      }
24
25      @Override
26      public void onDestroy(){
27          Log.d(TAG, msg: "onDestroy: ");
28      }
29
30  }
```

Сервис мы создали, manifestOn добавили. Теперь надо запустить MainActivity. Мы поступим следующим образом: создадим две кнопки. Одна будет запускать сервис, а другая – его останавливать.

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <LinearLayout
3       xmlns:android="http://schemas.android.com/apk/res/android"
4       xmlns:app="http://schemas.android.com/apk/res-auto"
5       xmlns:tools="http://schemas.android.com/tools"
6       android:layout_width="match_parent"
7       android:layout_height="match_parent"
8       android:orientation="vertical"
9       tools:context="elegion.com.servicereceivernotification.MainActivity"/>
10
11  <Button
12      android:id="@+id/btn_start_service"
```

```
13      android:text="Start_service"
14      android:layout_width="match_parent"
15      android:layout_width="wrap_content" />
16
17  <Button
18      android:id="@+id/btn_stop_service"
19      android:text="Stop Service"
20      android:layout_width="match_parent"
21      android:layout_width="wrap_content" />
```

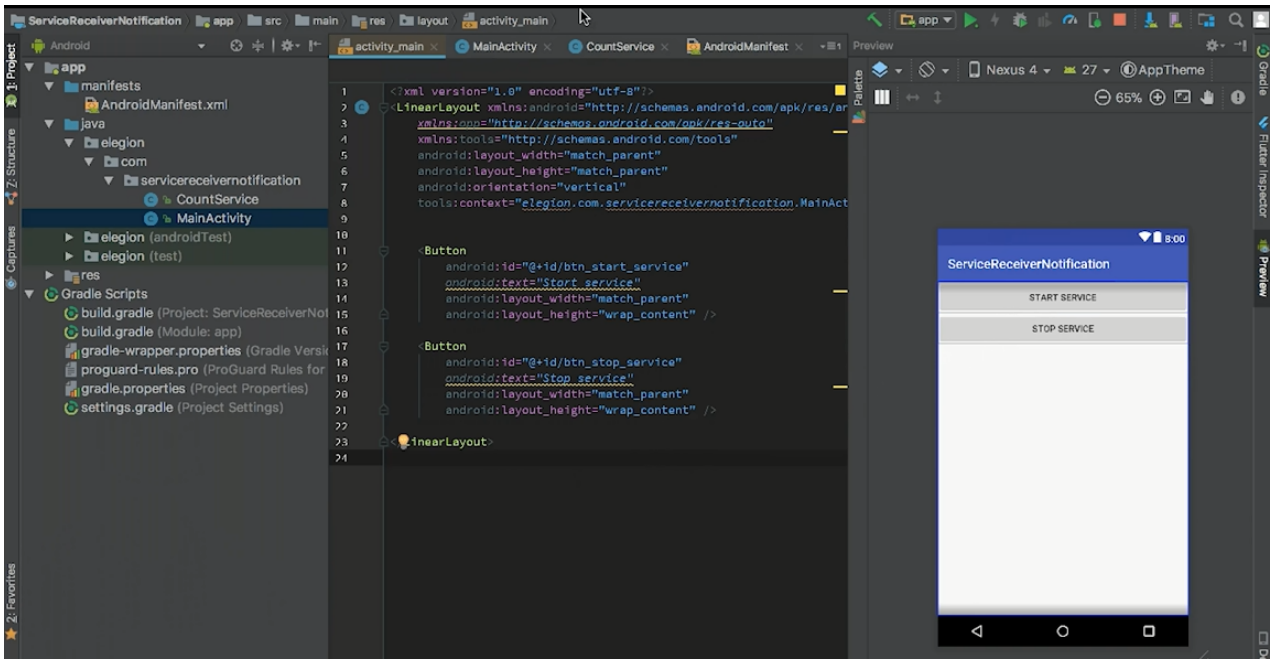



Рис. 1.5: Работа с приложением

Теперь нужно запустить сервис. В файле MainActivity, в методе onCreate() пишем:

```
1  package elegion.com.servicereceivernotification;
2
3  import...
4
```

```
5  public class MainActivity extends AppCompatActivity {
6
7      private Button mStartServiceBtn;
8      private Button mStopServiceBtn;
9
10     @Override
11     protected void onCreate(Bundle savedInstanceState) {
12         super.onCreate(savedInstanceState);
13         setContentView(R.layout.activity_main);
14
15         mStartServiceBtn = findViewById(R.id.btn_start_service);
16         mStopServiceBtn = findViewById(R.id.btn_Stop_service);
17
18         mStartServiceBtn.setOnClickListener(new View.onClickListener() {
19             @Override
20             public void onClick(View v) {
21                 Intent intent = new Intent(packageContext: MainActivity.this,
                    ↪  CountService.class);
22                 startService(intent);
23             }
24         }
25
26         mStopServiceBtn.setOnClickListener(new View.onClickListener() {
27             @Override
28             public void onClick(View v) {
29                 Intent intent = new Intent(packageContext: MainActivity.this,
                    ↪  CountService.class);
30                 stopService(intent);
31             }
32         }
33     }
34 }
```

Запустим наше приложение. Откроем log, уровень Debug. Щелкнем по кнопке «Start Service» и видим, что сработали методы onCreate() и onStartCommand(), это залогировано. Если мы еще раз щелкнем по кнопке «Start Service», то мы увидим, что onCreate() больше не вызывается, а вызывается только onStartCommand(), то есть запущенный сервис во второй раз создаваться не будет. При нажатии «Stop Service», как можно заметить, срабатывает onDestroy(). При повторном нажатии «Stop Service» ничего не происходит – остановить несуществующий сервис мы не

можем.

Нужно добавить задачу. Добавим ее в Count – сделаем так, чтобы он считал что-нибудь. Для этих целей нам подойдет ScheduledExecutor. Допишем его в метод onCreate() и запустим в методе onStartCommand().

```
1       import...
2           public class CountService extends Service {
3               public static final String TAG = CountService.class.getSimpleName();
4               private ScheduledExecutorService mScheduledExecutorService;
5
6               public CountService() {
7               }
8           }
9
10          @Override
11          public IBinder onBind(Intent intent){
12              return null;
13          }
14
15          @Override
16          public void onCreate(){
17              Log.d(TAG, msg: "onCreate: ");
18              mscheduledExecutorService = Executors.newScheduledThreadPool(corePoolSize:
                ↪  1);
19          }
20
21          @Override
22          public int onStartCommand(Intent intent, int flags, int startId){
23               Log.d(TAG, msg: "onStartCommand: ");
24               mScheduledExecutorService.scheduleΦeAtFixedRate(new Runnable() {
25                  @Override
26                  public void run() {
27                      Log.d(TAG, msg: "run: " + System.currentTimeMillis());
28                  }
29               }, InitialDelay: 1000, period: 1000, TimeUnit.MILLISECONDS);
30
31               return START_STUCKY;
32          }
33
```

```
34          @Override
35          public void onDestroy(){
36              Log.d(TAG, msg: "onDestroy: ");
37               mScheduledExecutorService.shutdownNow();
38          }
39
40      }
```

Поскольку в onStartCommand() мы выставили задержку и период в 1000 мс, то каждую секунду log будет выдавать нам текущие миллисекунды. Снова запускаемся, щелкаем на «Start Service». И в onDestroy() добавим shutdownNow, иначе ExecutorService будет работать даже после отключения от самого сервиса, так как он работает в отдельном потоке.

Таким нехитрым образом мы создали сервис и добавили в него периодическую таску. Еще раз напомню, что обычный сервис запускается в главном потоке, и чтобы он мог выполнять продолжительные задачи, какие-то тяжелые, который могут влиять на главный поток, обязательно нужно создать отдельный поток в сервисе, либо пользоваться intent-сервисом.

### 1.2.3. BroadcastReceiver, знакомство

В этом разделе мы познакомимся с очередным компонентом android-фреймворка – BroadcastReceiver. **BroadcastReceiver** – эир приемник сообщений, посылаемых системой либо другими приложениями при каком-либо событии, реагирующий на них и выполняющий какую-либо работу, отдельно от обычного процесса взаимодействия с приложением.

Приведем простой пример: вы открываете музыкальное приложение, подключаете наушники и включаете какую-нибудь композицию. Теперь, если вы выдернете наушники из устройства, музыка остановится. Откуда приложение знает, что наушники были выдернуты? Все просто: при подключении или отключении наушников система отправляет event «ACTION_HEADSET_PLUG», а приложение его обрабатывает, приостанавливая музыку. Подобных систем сообщений достаточно много.

При создании приемника мы наследуемся от базового класса BroadcastReceiver. В нем мы переопределяем метод onReceive(), добавляя нужную нам логику.

```java
public class MyReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent){
        //TODO: do something
    }
}
```

Однако чтобы приемник начал обрабатывать сообщения, его надо зарегистрировать. Возможны два варианта регистрации приемника:

- В манифесте приложения. Приемник, зарегистрированный в манифесте, будет принимать сообщения даже если само приложение не запущено на момент срабатывания события. В этом его главное отличие от приемника, зарегистрированного в коде.

```xml
<receiver

 android:name=".MyReceiver"
 android:enabled="true"
 android:exported="true">
 <intent-filter>

 <action android:name="android.intent.action.HEADEST_PLUG"/>
 </intent-filter>
</receiver>
```

- В коде. Для регистрации приемника таким образом используется метод registerReceiver(). Соответственно, чтобы отвязать приемник, используется метод unRegisterReceiver(). Конечно же, мы не ограничены только системными сообщениями: мы можем отправлять свои. Для этого используется метод sendBroadcast(), в аргументе которого передается intent с нужным нам action'ом, в который и подписан приемник. Также имеет смысл передавать какие-либо данные через putExtra().

```java
MyReceiver myReceiver = new MyReceiver();
IntentFilter = new IntentFilter(Intent.ACTION_HEADSET_PLUG);
registerReceiver(myReceiver, filter);
```

В Android также доступны следующие возможности. Например, мы можем использовать метод sendOrderedBroadcast. И если несколько приемников подписаны на один и тот же action, то они будут получать сообщения согласно своему приоритету, по очереди. Каждый приемник сможет получить сообщение, обработать и передать его дальше, либо вообще не передавать.

Приведем пример. Допустим, у нас приложение-мессенджер и есть два приемника, настроенных получать push-уведомления сервера о новых сообщениях. Один из приемников – с низким приоритетом – зарегистрирован в манифесте. Работа его заключается в том, что он показывает уведомления о новом сообщении, как только получает свое событие. Пользователь будет знать о новых сообщениях даже если приложение в данный момент не работает. Другой приемник зарегистрирован в коде, у него более высокий приоритет. Мы предполагаем, что пользователю не нужно уведомление о сообщении, если он и так в данный момент находится в приложении и своими глазами видит это новое сообщение. Поэтому наш приемник, зарегистрированный в коде, гасит event и не передает его дальше приемнику, зарегистрированному в манифесте.

```
1  Intent intent = new Intent();
2  intent.setAction("com.e_legion.coursera.SOMETHING_JUST_HAPPEN");
3  Intent.putExtra("ARG_DATA", "SOME_VALUE");
4  sendBroadcast(intent);
```

Метод sendBroadcast() передает сообщение всей системе. В качестве альтернативы можно использовать методы LocalBroadcastManager.sendBroadcast() для отправки сообщения и LocalBroadcastManager.registerReceiver() для регистрации приемника. При использовании менеджера, сообщения будут отправляться только в пределах приложения. Если у вас нет необходимости, чтобы система или другие приложения знали о ваших сообщениях и реагировали на них, безопаснее и производительнее всего будет использовать этот способ.

При использовании BroadcastReceiver нужно иметь в виду некоторые ограничения. BroadcastReceiver живет до тех пор, пока выполняется код в методе onReceive(). Этот метод работает в главном потоке приложения и не подходит для каких-либо долгих операций, так как будет прибит системой через 5-10 секунд. Если нужно выполнить такую операцию, то необходимо запускать сервис из приемника. Также чтобы реагировать на некоторые системные события, нужны разрешения, которые также прописываются в манифесте.

### 1.2.4. Создание BroadcastReceiver

В этом разделе мы продолжим разбираться с сервисами: мы создадим и зарегистрируем, поймаем системное событие, поймаем свое событие уже в Receiver'е.

Щелкаем New –> Java class. Name: SimpleReceiver. Superclass: android.content.BroadcastReceiver. Go! Нажимаем Alt+Enter, Implement methods, onReceive(). Что делает Receiver? Он принимает intent, если у него совпадает action. При этом он живет и работает, пока существует метод onReceive(). Поскольку создание Receiver'а, происходящее в методе onReceive(), работает в главном потоке, сильно его загромождать нельзя. Конкретно этот Receiver будет просто показывать action, который он поймал.

```
1  package elegion.com.servicereceivernotification;
2
3  import android.content.BroadcastReceiver;
4  import android.content.Context;
5  import android.content.Intent;
6  import android.widget.Toast;
7
8  public class SimpleReceiver extends BroadcastReceiver {
9      @Override
10     public void onReceive(Context context, Intent intent) {
11         Toast.makeText(context, intent.getAction(), Toast.LENGTH_SHORT).show();
12     }
13 }
```

Далее зарегистрируем Receiver. Как мы уже знаем, сделать это можно двумя способами: динамически (будет работать, пока работает activity) или статически, в манифесте, и тогда он сработает, даже если приложение не запущено. Этот случай мы обсудим в конце чуть подробнее. В методе onCreate() создаем SimpleReceiver. Регистрировать Receiver будем в методе onResume(). Убирать регистрацию будем в методе onPause(). Так как Receiver – не effected интерфейс, не имеет смысла регистрировать его в onStart(). При регистрации в качестве аргумента мы передаем сам receiver и intent-filter – фильтр тех intent'ов, которые он будет ловить. Не забываем убрать регистрацию в onResume().

```
1  package elegion.com.servicereceivernotification;
2
3  import...
4
```

```
5  public class MainActivity extends AppCompatActivity {
6
7      private Button mStartServiceBtn;
8      private Button mStopServiceBtn;
9      private SimpleReceiver mSimpleReceiver;
10     private IntentFilter mIntentFilter;
11
12     @Override
13     protected void onCreate(Bundle savedInstanceState) {
14         super.onCreate(savedInstanceState);
15         setContentView(R.layout.activity_main);
16
17         mStartServiceBtn = findViewById(R.id.btn_start_service);
18         mStopServiceBtn = findViewById(R.id.btn_Stop_service);
19
20         mStartServiceBtn.setOnClickListener(new View.onClickListener() {
21             @Override
22             public void onClick(View v) {
23                 Intent intent = new Intent(packageContext: MainActivity.this,
                    ↪  CountService.class);
24                 startService(intent);
25             }
26         }
27
28         mStopServiceBtn.setOnClickListener(new View.onClickListener() {
29             @Override
30             public void onClick(View v) {
31                 Intent intent = new Intent(packageContext: MainActivity.this,
                    ↪  CountService.class);
32                 stopService(intent);
33             }
34         }
35
36         mSimpleReceiver = new SimpleReceiver();
37         mIntentFilter = new IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED);
38     }
39
40     @Override
41     protected void onResume() {
42         super.onResume();
```

```
43          registerReceiver(mSimpleRecevier, intentFilter);
44      }
45
46      @Override
47      protected void onPause() {
48          super.onPause();
49          unregisterReceiver(mSimpleRecevier);
50      }
51  }
```

Итак, мы создали Receiver и зарегистрировали его на ловлю состояния режима «В самолете». Запускаем приложение и приложение и смотрим, что получилось. Щелкаем на режим «В самолете» и видим Toast, вызванный из Receiver. Щелкаем еще раз – еще раз видим Toast, то есть при изменении режима Receiver срабатывает. Что, если я сверну приложение и попытаюсь снова включить режим «В самолете»? Ничего не происходит. Это связано с тем, что в методе onPause() мы Receiver отвязали. Таким образом, мы поймали системное событие. Теперь поймаем свое событие. Ловля своего события принципиально не отличается от предыдущего случая. Отличие лишь в том, что его нужно отправить.

Для отправки события создадим еще одну кнопку.

```
1  <?xml version="1.0" encoding="utf-8"?>
2
3    ...
4
5  <Button
6      android:id="@+id/btn_send_broadcast"
7      android:text="Send Broadcast"
8      android:layout_width="match_parent"
9      android:layout_width="wrap_content" />
```

Переходим в Activity, добавляем кнопку. Отправляем Broadcast. Сделать это можно двумя способами: либо всей системе методом sendBroadcast(), либо локально приложению методом LocalBroadcastManager.sendBroadcast(). Мы воспользуемся первым методом, так как второй связан с созданием дополнительных объектов. В качестве аргумента sendBroadcast() будет принимать intent, причем делать это он будет неявно.

```
package elegion.com.servicereceivernotification;

import...

public class MainActivity extends AppCompatActivity {

    private Button mStartServiceBtn;
    private Button mStopServiceBtn;
    private Button mSendBroadcastBtn;
    ...

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

        mSendBroadcastBtn.setOnClickListener(new View.onClickListener() {
            @Override
            public void onClick(View v) {
                sendBroadcast(!!!!!!!!!!!);
            }
        }
        ...
    }
    ...

    }
}
```

В самом Receiver создаем psfs. Кастомные события, кастомные action'ы должны быть как можно более уникальными, поэтому хорошим подходом является использование в качестве префикса action'а названия пакета. В нашем случае это "elegion.com.servicereceivernotification".

```
package elegion.com.servicereceivernotification;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.widget.Toast;

```

```
8  public class SimpleReceiver extends BroadcastReceiver {
9
10     public static final String SIMPLE_ACTION =
       ↪  "elegion.com.servicereceivernotification.SIMPLE_ACTION";
11     @Override
12     public void onReceive(Context context, Intent intent) {
13         Toast.makeText(context, intent.getAction(), Toast.LENGTH_SHORT).show();
14     }
15 }
```

Далее возвращаемся обратно в MainActivity и передаем в sendBroadcast новый Intent. В качестве action'а передаем тот самый Receiver, который мы создали. И в IntentFilter сообщаем, что мы теперь ловим собственное сообщение. Получается, что по кнопке мы передаем broadcast с action'ом, в Receiver'е в IntentFilter ловим этот action. Регистрируем Receiver в onResume(), убираем регистрацию в onPause(), как и раньше, и при ловле action'а показываем Toast.

```
1  package elegion.com.servicereceivernotification;
2
3  import...
4
5  public class MainActivity extends AppCompatActivity {
6
7      private Button mStartServiceBtn;
8      private Button mStopServiceBtn;
9      private Button mSendBroadcastBtn;
10     ...
11
12     @Override
13     protected void onCreate(Bundle savedInstanceState) {
14         ...
15
16         mSendBroadcastBtn.setOnClickListener(new View.onClickListener() {
17             @Override
18             public void onClick(View v) {
19                 sendBroadcast(new Intent(SimpleReceiver.SIMPLE_ACTION));
20             }
21         }
22
23         mSimpleReceiver = new SimpleReceiver();
```

```
24              mIntentFilter = new IntentFilter(Intent.SIMPLE_ACTION);
25          }
26          ...
27          }
28      }
```

Запускаем приложение, щелкаем по «Send Broadcast» и видим action, который мы создавали.

### 1.2.5. Связка Activity-Service-BroadcastReceiver-Activity

В этом видео мы разберемся, как ловить события сервиса. Спойлер: это делается очень просто. Скопируем строчку sendBroadcast(new Intent(SimpleReceiver.SIMPLE_ACTION));, перейдем в CountService и вставим ее в конец метода run(). Сменим период на 4000 миллисекунд вместо 1000, потому что в Receiver'е мы ловим Intent и показываем короткий Toast, который длится 2.5 секунды. Поэтому если мы будем отправлять broadcast каждые 1000 миллисекунд, то Toast просто не будет успевать показываться.

Запускаем сервис. Нажимаем «Start Service», видим Toast, который показывается каждые 2.5 секунды. Останавливаем сервис.

Давайте попробуем переслать не просто Intent, а какие-то данные из сервиса в BroadcastReceiver. В методе run():

```
1   import...
2       public class CountService extends Service {
3           public static final String TAG = CountService.class.getSimpleName();
4           public static final String TIME = "TIME";
5
6           private ScheduledExecutorService mScheduledExecutorService;
7
8           public CountService() {
9           }
10      }
11      ...
12      @Override
13      public int onStartCommand(Intent intent, int flags, int startId){
```

```
14            Log.d(TAG, msg: "onStartCommand: ");
15            mScheduledExecutorService.scheduleAtFixedRate(new Runnable() {
16                @Override
17                public void run() {
18                    long currentTimeMillis = System.currentTimeMillis();
19                    Log.d(TAG, msg: "run: " + currentTimeMillis);
20                    Intent intentToSend = new Intent(SimpleReceiver.SIMPLE_ACTION);
21                    intentToSend.putExtra(TIME, currentTimeMillis);
22                    sendBroadcast(intentToSend);
23                }
24            }, InitialDelay: 1000, period: 4000, TimeUnit.MILLISECONDS);
25
26            return START_STUCKY;
27        }
28        ...
29    }
```

Что мы написали: мы каждые 4000 мс создаем intent, оборачиваем туда текущее значение миллисекунд и отправляем его в broadcast'е. Соответственно, в самом BroadcastReceiver'е мы должны эти данные извлечь. В методе onReceive() мы получаем intent на вход. В качестве Toast'а показываем строку со временем: "current time is " + time.

```
1   ...
2
3   @Override
4   public void onReceive(Context context, Intent intent) {
5       long time = intent.getLongExtra(CountReceiver.TIME, defaultValue: 0L);
6       Toast.makeText(context, text: "current time is " + time,
        ↪  Toast.LENGTH_SHORT).show();
7   }
```

Запускаем приложение, запускаем сервис кнопкой «Start Service», видим наш Toast со временем. Таким образом, мы передали данные из сервиса в BroadcastReceiver.

Теперь попробуем передать данные из BroadcastReceiver'а в activity. Один из самых простых вариантов, как можно это сделать, заключается в запуске activity в самом BroadcastReceiver'е. Для этого через стандартный генератор создадим пустой activity:

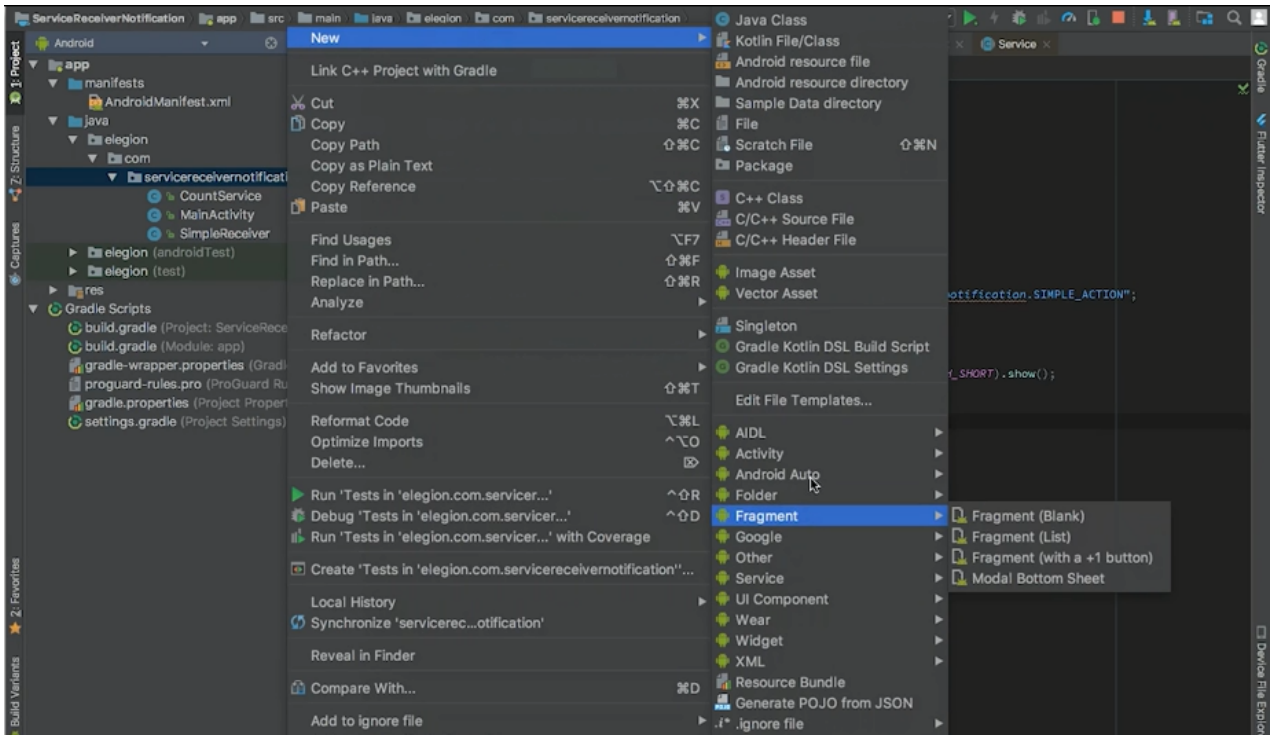



Рис. 1.6: Создание нового активити

Назовем его TempActivity.

```
1  package elegion.com.servicereceivernotification;
2
3  import...
4
5  public class TempActivity extends AppCompatActivity {
6
7      @Override
8      protected void onCreate(Bundle savedInstanceState) {
9          super.onCreate(savedInstanceState);
10         setContentView(R.layout.activity_temp);
11     }
12 }
```

Кликаем на activity_temp:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    tools:context="elegion.com.servicereceivernotification.TempActivity">

    <TextView
        android:id="@+id/tv_time"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

</LinearLayout>
```

Переключаемся обратно в наш Receiver и в нем запускаем activity. Для этого нужно создать новый Intent. И давайте также перенесем данные.

```
...

@Override
public void onReceive(Context context, Intent intent) {
    long time = intent.getLongExtra(CountReceiver.TIME, defaultValue: 0L);
    Toast.makeText(context, text: "current time is " + time,
     ↪  Toast.LENGTH_SHORT).show();

    Intent launchActivityIntent = new Intent(context, TempActivity.class);
    launchActivityIntent.putExtra(CountReceiver.TIME, time);
    context.startActivity(launchActivityIntent);
}
```

А в самом TempActivity напишем:

```
package elegion.com.servicereceivernotification;

import...

public class TempActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_temp);
        long longExtra = getIntent().getLongExtra(CountReceiver.TIME, defaultValue:
        ↪  0L);
        TextView tvTime = findViewById(R.id.tv_time);
        tvTime.setText("time is " + longExtra);
    }
}
```

Посмотрим, что мы сделали. В CountService мы передаем в intent'е, который мы передаем в broadcastReceiver, мы передаем текущее время. В Receiver'е мы это время ловим, показываем в Toast'е. Создаем новый Intent, загоняем туда данные – текущее время – и запускаем новый Activity. В новом Activity мы показываем полученное время в TextView.

Запускаем сервис, запустилось новое Activity – и на этом все, потому что, поскольку мы переключились на новое Activity, Receiver больше не работает, ведь он остался в старом Activity. Грусть, печаль, тоска. В силу периодичности, мы видим сменяющиеся экраны выбора Start/Stop Service и нового Activity, причем Toast показывает разное время, значит, программа работает так, как и задумывалось.

Теперь поймаем событие, выпущенное нашим текущим Activity – тем, в котором мы его зарегистрировали. Для этого есть способ, но он весьма изворотлив. Перейдем в MainActivity и создадим TextView. Передадим этот TextView в сам Receiver.

```
<?xml version="1.0" encoding="utf-8"?>

<TextView
        android:id="@+id/tv_time"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
```

В файле SimpleReceiver создадим конструктор, который на вход принимает TextView. Закомментируем часть кода, которая отвечает за запуск нового activity.

```
1  package elegion.com.servicereceivernotification;
2
3  import android.content.BroadcastReceiver;
4  import android.content.Context;
5  import android.content.Intent;
6  import android.widget.TextVie;
7  import android.widget.Toast;
8
9  public class SimpleReceiver extends BroadcastReceiver {
10
11     private weakReference<TextView> mTextViewWeakReference;
12
13     public SimpleReceiver(TextView textView){
14         mTextViewWeakReference = new weakReference<TextView>(TextView) ;
15
16     }
17
18     public static final String SIMPLE_ACTION =
       ↪  "elegion.com.servicereceivernotification.SIMPLE_ACTION";
19     @Override
20     public void onReceive(Context context, Intent intent) {
21         long time = intent.getLongExtra(CountReceiver.TIME, defaultValue: 0L);
22         Toast.makeText(context, text: "current time is " + time,
           ↪  Toast.LENGTH_SHORT).show();
23
24         TextView textView = mTextViewWeakReference.get();
25         StringBuilder builder = new StringBuilder(textView.getText().toString());
26         builder.append(time).append("\backslash n");
27         textView.setText(builder.toString());
28
29         // Intent launchActivityIntent = new Intent(context, TempActivity.class);
30         // launchActivityIntent.putExtra(CountReceiver.TIME, time);
31         // context.startActivity(launchActivityIntent);
32     }
33 }
```

Возвращаемся в MainActivity и дописываем, что при создании SimpleReceiver мы будем прини-

мать TextView – приведенный к этому типу findViewById(R.id.tv_time).

```
package elegion.com.servicereceivernotification;

import...

public class MainActivity extends AppCompatActivity {

    ...

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

        ...

        mSimpleReceiver = new SimpleReceiver((TextView) findViewById(R.id.tv_time));
        mIntentFilter = new IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED);
    }

    ...
}
```

Запускаем, щелкаем на «Start Service» – работает. Мы видим, что мы запускаем сервис, в нем получаем новые данные, их оборачиваем в intent, отправляем в BroadcastReceiver, а он, в свою очередь, достает эти данные и записывает их в TextView, который он получил при создании в конструкторе и который он хранит в слабой ссылке. Это дичь. Это ужас. Так делать нельзя.

Для взаимодействия с интерфейсом сервисы, пусть через BroadcastReceiver, не очень хорошо подходят. То есть это можно сделать, как мы только что сделали, но это неправильно. Если вам нужно, чтобы ваш сервис взаимодействовал с activity, чтобы он отправлял запросы и получал результаты, нам нужно не только запустить сервис, но и привязать его. То есть вместо StartService() вызывать BindService(). Эта тема достаточно интересна, поэтому мы оставляем ее читателям на самостоятельное изучение.

Таким образом, в этом разделе мы увидели слабое место связки Activity-Service-BroadcastReceiver-Activity: работа осложнена тем, что нет прямых ссылок на View-элементы.

## 1.2.6. PendingIntent, Notification, NotificationManager

### Для чего?

Иногда бывают случаи, когда необходимо оповестить пользователя о том, что в нашем приложении что-то случилось, например, появилось новое предложение или акция и т.д., но при этом наше приложение закрыто или телефон вообще лежит где-нибудь на полке. Для этого существует специальный механизм для отображения уведомлений, который использует PendingIntent, Notification, NotificationManager. Давайте разберемся, что же это такое и как оно должно работать.

### PendingIntent. Для чего?

Вы уже знакомы с Intent, так что же такое PendingIntent? PendingIntent – обертка, которая позволяет стороннему приложению выполнять определенный код из вашего приложения с правами, которые определены для вашего же приложения.

Если в стороннее приложение передать простой Intent, то он будет выполняться с теми правами, которые имеет само стороннее приложение.

Т.е. PendingIntent позволяет стороннему приложению, в которое его передали, запустить хранящийся внутри него Intent, от имени того приложения, и теми же с полномочиями, передавшего этот PendingIntent. А Intent работает внутри приложения.

Звучит сложно, но достаточно запомнить следующее: для того, чтобы система (другое приложение) могло запустить ваше, ему нужно передать PendingIntent.

Дополнительная информация:

1. http://startandroid.ru/ru/uroki/vse-uroki-spiskom/204-urok-119-pendingintent-flagi-requestcode-alarmmanager.html
2. http://startandroid.ru/ru/uroki/vse-uroki-spiskom/160-urok-95-service-obratnaja-svjaz-s-pomoschju-pendingintent.html
3. https://developer.android.com/reference/android/app/PendingIntent.html
4. http://codetheory.in/android-pending-intents/

### Notification, NotificationManager. Для чего?

Notification – нужен, чтобы задать свойства того, как будет выглядеть наше уведомление в строке состояния, какой у неё будет звук, сообщение и т.д.

NotificationManager – нужен, чтобы управлять всеми уведомлениями нашего приложения. NotificationManager создается при помощи вызова метода getSystemService() или метода from(), а когда надо показать уведомление пользователю, вызывается метод notify().

Эти сущности служат для того, чтобы выводить уведомления в строке состояний (status bar).


### PendingIntent, Notification, NotificationManager. Как работает?

Сейчас вы узнаете, как можно отобразить уведомление.

Объект **Notification** должен содержать следующие элементы:

- setSmallIcon() – задаем небольшой значок;
- setContentTitle() – задаем небольшой заголовок;
- setContentText() – задаем подробный текст.

Пример показа уведомления:

```
1  Intent notificationIntent = new Intent(this, MainActivity.class);
2  PendingIntent contentIntent = PendingIntent.getActivity(this,
3       0, notificationIntent,
4       PendingIntent.FLAG_CANCEL_CURRENT);
5
6  // до версии Android 8.0 API 26
7  NotificationCompat.Builder builder = new NotificationCompat.Builder(this);
8
```

```
builder.setContentIntent(contentIntent)
     // обязательные настройки
     .setSmallIcon(R.drawable.ic_android_black_24dp)
     .setContentTitle("Заголовок")
     .setContentText("Текст уведомления")
     // некоторые необязательные настройки
     .setLargeIcon(BitmapFactory.decodeResource(getResources(),
     ↪  R.mipmap.ic_launcher_round)) // большая иконка
     .setTicker("Текст в строке состояния")
     .setWhen(System.currentTimeMillis()) // время, которое отобразится в уведомлении
     .setAutoCancel(true); // автоматически закрыть уведомление после нажатия

// с помощью getSystemService
//NotificationManager notificationManager = (NotificationManager)
 ↪  getSystemService(Context.NOTIFICATION_SERVICE);
// с помощью from
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(123, builder.build());
```

С другими параметрами и возможностями уведомлений вы можете ознакомиться здесь:

1. https://developer.android.com/guide/topics/ui/notifiers/notifications.html?hl=ru
2. http://developer.alexanderklimov.ru/android/notification.php

### Notification Channels. Что это и для чего?

В Android O появились каналы уведомлений, представляющие собой некие группировки по каналам, позволяющие пользователям применять разом для всех уведомлений канала соответствующие настройки. Например, полностью блокировать канал или изменить уровень важности. Сразу поясним, что такое уровень важности. Начиная, с вышеупомянутой версии уровни очередности больше не работают для конкретных уведомлений, вместо этого при создании канала, вы задаете уровень важности уведомлений – от NotificationManager.IMPORTANCE_NONE до NotificationManager.IMPORTANCE_HIGH. Помимо этого мы можем определить цвет уведомления, звук, наличие вибрации и возможность отображения на заблокированном экране.

```
int importance = NotificationManager.IMPORTANCE_LOW;
//Задаём уровень важности
NotificationChannel notificationChannel = new NotificationChannel(channelId,
    channelName, importance);
//Создаём канал
notificationChannel.enableLights(true);
//Даём возможность изменить цвет
notificationChannel.setLightColor(Color.RED);
//И выбираем его
notificationChannel.enableVibration(true);
notificationChannel.setVibrationPattern(new long[]{100, 200, 300, 400, 500, 400, 300,
    200, 400});
//Разрешаем вибрацию и задаём её последовательность.
```

Исходя из этого в Android O при возвращении объекта из билдера нотификации требуется указать id соответствующего ему канала при помощи метода setChannel(). Это делается чтобы убедиться, что пользователь может применять настройки для него через канал, к которому он относится и кастомизируется в сеттингах девайса.

Также стоит отметить, что канал может быть удален следующим образом.

```
notificationManager.getNotificationChannel("some_channel_id");
notificationManager.deleteNotificationChannel(notificationChannel);
```

Если вы создадите новый канал с этим же идентификатором, удаленный канал будет воссоздан с набором тех настроек, которые были у него до его удаления.

Дополнительная информация:

1. https://medium.com/exploring-android/exploring-android-o-notification-channels-94cd274f604c

### 1.2.7. Показ Notification

В прошлом разделе мы научились ловить события из сервиса и показывать их на интерфейсе. В этой части курса мы научимся создавать уведомления, чтобы сделать наш сервис foreground'ным.

Первый вопрос, который возникает: зачем вообще делать сервис foregroung'ным? Ведь на текущем этапе у нас есть работающий сервис и с ним все хорошо. Если мы посмотрим в Logcat, в котором показывается работа сервиса, можно увидеть событие run() и некоторые другие. Давайте свернем наше приложение и откроем другое, например, Google-карты. Будем искать в нем какие-нибудь города, например, Лондон. Щелкнем на поиск пути и посмотрим на дорогу в Лондон из Парижа. Маршрут занимает 2 часа и 35 минут на поезде EuroStar. А что при этом делает сервис? Logcat подтверждает, что сервис при этом продолжает работать. Но лишь до некоторого момента.

Когда мы переключаемся на другое приложение и оно в foreground'е, у нас на экране система старается перераспределить ресурсы (память, процессорное время и т.п.) так, чтобы оби доставались именно этому приложению. Делается это для того, чтобы у пользователя был хороший опыт, чтобы он не ругался на то, что android-телефоны тормозят.

Мы запустили сервис и переключились на другое приложение. Система в какой-то момент начала понимать, что ей не хватает памяти на это приложение. И она начала искать, что бы такого можно было прибить, чтобы освободить ресурсы. И ее выбор пал на наш сервис, который в тот момент работал – считал миллисекунды. Система поняла, что несмотря на то, что сервис работает, пользователь результат работы не видит. И из этого система сделала вывод, что результаты и сервис не нужны и не важны и приняла решение прихлопнуть данный сервис, что мы и видим в Logcat'е – сработал метод onDestroy().

И именно чтобы система не убивала наши сервисы, нужно сделать его foreground, то есть показать пользователю, что в фоне что-то происходит. Чтобы сделать сервис таким, нам нужно просто вызвать метод startForeground, принимающий на вход id – канал, в который уведомление будет попадать, и, собственно, само уведомление. Если мы отправим несколько уведомлений с одинаковым id, то новые уведомления будут менять content старого уведомления.

```
import...
    public class CountService extends Service {
        public static final String TAG = CountService.class.getSimpleName();
        public static final String TIME = "TIME";

        private ScheduledExecutorService mScheduledExecutorService;

        public CountService() {
        }
    }
...
```

```
12      @Override
13      public int onStartCommand(Intent intent, int flags, int startId){
14          Log.d(TAG, msg: "onStartCommand: ");
15
16          startForeground(id: 123, !!!!!)
17
18          mScheduledExecutorService.scheduleAtFixedRate(new Runnable() {
19              @Override
20              public void run() {
21                  ...
22              }
23          }, InitialDelay: 1000, period: 4000, TimeUnit.MILLISECONDS);
24
25          return START_STUCKY;
26      }
27      ...
28  }
```

Создадим в файле CountService методы, один из которых будет создавать builder, а другой – непосредственно сами уведомления. Для начала в методе onCreate инициализируем instanceNotification менеджера, не забывая при этом откастовать. В зависимости от того, какого уровня API, мы создаем NotificationBuilder либо с, либо без канала. В Android 26-го уровня появились NotificationChannel'ы. И если мы таргетимся на Oreo и выше, нам обязательно нужно уведомление загнать в канал, в котором оно и будет показываться. В случае, если канал еще не существует, его нужно создать и добавить(см. else). В getNotificationBuilder() используется такой параметр, как IMPORTANCE – важность уведомления. В зависимости от этого параметра, уведомление будет отрабатывать со звуком или без звука, с вибрацией или без и так далее. Чтобы не создавать канал каждый раз ( createNotificationChannel(channel);), удостоверимся в его отсутствии.

```
1   import...
2       public class CountService extends Service {
3           public static final String TAG = CountService.class.getSimpleName();
4           public static final String TIME = "TIME";
5
6           private ScheduledExecutorService mScheduledExecutorService;
7           private NotificationManager mManager;
8           private NotificationCompat.Builder mBuilder;
9           public CountService() {
10          }
```

```
11
12          ...
13
14          @Override
15          public void onCreate(){
16              Log.d(TAG, msg: "onCreate: ");
17              mscheduledExecutorService = Executors.newScheduledThreadPool(corePoolSize:
                ↪  1);
18
19              mManager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
20              mBuilder = getNotificationBuilder();
21
22
23          }
24
25          private NotificationCompat.Builder getNotificationBuilder() {
26              if(Build.VERSION.SDK_INT < Build.VERSION_CODES.O) {
27                  return new NotificationCompat.Builder(context: this)
28                  } else {
29                      String channel_id = "my_channel_id";
30
31                      if(mManager.getNotificationChannel(channelId) == null) {
32                          NotificationChannel channel = new
                            ↪  NotificationChannel(channel_id, name: "Text for user",
                            ↪  NotificationManager.IMPORTANCE_LOW);
33                          mManager createNotificationChannel(channel);
34                      }
35
36
37                      return new NotificationCompat.Builder(context: this, channelId:
                        ↪  channel_id)
38                  }
39              }
40
41          };
42          ...
43  }
```

Теперь нам нужно создать подложку – внешний вид нашего уведомления. И мы к тому же хотим,

чтобы наши текущие миллисекунды попадали в текст нашего текущего уведомления. Для этого в методе onCreate() и onStartCommand():

```
@Override
public void onCreate(){
    Log.d(TAG, msg: "onCreate: ");
    mscheduledExecutorService = Executors.newScheduledThreadPool(corePoolSize: 1);

    mManager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    mBuilder = getNotificationBuilder();

    mBuilder.setContentTitle("Count service notifications");
            .setSmallIcon(R.drawable.ic_launcher_foreground);
}

...

@Override
    public int onStartCommand(Intent intent, int flags, int startId){
         Log.d(TAG, msg: "onStartCommand: ");

         startForeground(id: 123, getNotification(contentText: "start notification"));

         mScheduledExecutorService.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                long currentTimeMillis = System.currentTimeMillis();
                Log.d(TAG, msg: "run: " + currentTimeMillis);

                mManager.notify(id: 123, getNotification(contentText: "time is " +
                 ↪  currentTimeMillis))
                Intent intentToSend = new Intent(SimpleReceiver.SIMPLE_ACTION);
                intentToSend.putExtra(TIME, currentTimeMillis);
                sendBroadcast(intentToSend);
            }
         }, InitialDelay: 1000, period: 4000, TimeUnit.MILLISECONDS);

         return START_STUCKY;
    }

```

```
37      private Notification getNotification(String contentText){
38          return mBuilder.setContentText(contentText).build();
39      }
```

Запустим наш сервис и видим уведомление, в котором отображается текущее время в миллисекундах. Таким образом, мы создали foreground service, создали уведомление, способное к обновлению. Снова откроем изначальное приложение с картами Google и найдем поезда из Москвы в Санкт-Петербург. А теперь снова посмотрим в Logcat и видим, что сервис действительно продолжает свою работу, так как он foreground'ный. Он работает в нотификации, поэтому я как пользователь вижу, что у меня что-то происходит и я не буду ругаться, если у меня вдруг внезапно истощится батарейка. Попробуем добавить немного экшна. В getNotification():

```
1   private Notification getNotification(String contentText){
2       Intent intent = new Intent(packageContext: this, TempActivity.class);
3       intent.putExtra(TIME, contentText);
4       PendingIntent pendingIntent = PendingIntent.getActivity(context: this,
        ↪  requestCode: 2434, intent, PendingIntent.FLAG_UPDATE_CURRENT);
5       return mBuilder.setContentText(contentText)
6                      .setContentIntent(PendingIntent);
7                      .build();
8   }
```

Переходим к TempActivity.

```
1   public class TempActivity extends AppCompatActivity {
2
3       @Override
4       protected void onCreate(Bundle savedInstanceState) {
5           super.onCreate(savedInstanceState);
6           setContentView(R.layout.activity_temp);
7
8           String StringExtra = getIntent().getStringExtra(CountReceiver.TIME,
            ↪  defaultValue: 0L);
9           TextView tvTime = findViewById(R.id.tv_time);
10          tvTime.setText("this is from notification" + stringExtra);
11      }
12  }
```

Запускаем сервис: «Start service», увидели нотификацию, щелкнули по ней и попали в TempActivity, сравнили времена – все действительно работает. Мы бы рассказали побольше про нотификации, но не можем этого сделать, потому что в каждом новом Android'е они меняются (так, например, в Android Oreo появились каналы). Поэтому нужно периодически подтверждать актуальность имеющейся у вас информации на https://developer.android.com/index.html.

### 1.2.8. BroadcastReceiver в манифесте

Этот раздел будет последним, посвященным сервисам, и на нем мы рассмотрим бонусную тему – статическое объявление Receiver'а в манифесте. Почему эта часть является бонусной, а не включена в основную программу? Дело в том, что Android все время развивается, меняется и вектор его развития направлен на то, чтобы сделать жизнь пользователей лучше. Но, как говорится, что пользователю хорошо, то разработчику боль, потому что хорошо пользователю делается путем стягивания гаек разработчику. И в последнем Android – Oreo – выкатили новое требование: нельзя декларировать в манифесте Receiver'ы. То есть декларировать можно только те Receiver'ы, которые имеют непосредственное отношение к вашему приложению, за несколькими исключениями. Так, в одном из предыдущих видео мы показывали реакцию на системное событие – изменение режима «В самолете». И динамический Receiver его ловил, а статический его уже не поймает, потому что мы показывали на эмуляторе с версией Oreo на борту.

Что предлагают взамен? Взамен предлагают воспользоваться другими способами понять, что же происходит в системе. Вместо того, чтобы ловить Intent, что вы подключились к вайфаю или что у вас садится батарейка, лучше воспользоваться специальными сервисами. Вместо этого предлагается пользоваться специальными сервисами – Job Schedule сервисами, которые запускаются в некоторый определенный момент или при определенных условиях. То есть с Job Schedule сервиса можно запуститься, если девайс подключен к зарядке, к вайфаю и пользователь нас не трогает.

Давайте создадим broadcastReceiver – StaticReceiver. Он добавился в манифест, однако в нем не прописан IntentFilter.

```
<receiver
     android:name=".StaticReceiver"
     android:enabled="true"
     android:exported="true">
     <intent-filter>
         <action android:name = "android.intent.action.AIRPLANE_MODE"/>
     </intent-filter>
```

```
</receiver>
```

Внутри StaticReceiver создаем intent:

```
public class StaticReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent){
        Intent launchIntent = new Intent(context, MainActivity.class);
        launchIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
        ↪  Intent.FLAG_ACTIVITY_CLEAR_TASK);
        context.startActivity(launchIntent);
    }
}
```

И запускаем. Мы создали эмулятор 22-го уровня (Lollipop). Убираем приложение из дока (нет запущенных приложений). Открываем панель уведомлений, нажимаем на иконку режима в самолете и тут запускается наше приложение – все работает именно так, как мы это задумывали.

В новых версиях Android нежелательно декларировать общие action'ы в манифесте. Исключения, конечно, есть, но в целом лучше декларировать только то, что относится к вашему приложению.

### 1.2.9. Материалы для самостоятельного изучения

1. https://developer.android.com/guide/components/services.html?hl=ru
2. https://developer.android.com/guide/components/bound-services.html?hl=ru
3. https://developer.android.com/guide/components/broadcasts.html?hl=ru
4. https://codetheory.in/understanding-android-started-bound-services/
5. https://developer.android.com/training/notify-user/build-notification.html
6. https://developer.android.com/guide/topics/ui/notifiers/notifications.html?hl=ru
7. https://code.tutsplus.com/ru/tutorials/android-o-how-to-use-notification-channels–cms-28616

# 1.3. Многопоточность в Android

## 1.3.1. AsyncTask, знакомство

Среди Android-разработчиков AsyncTask пользуется дурной славой: им пренебрегают, его недолюбливают и всячески избегают. Все из-за неправильного понимания того, как AsyncTask работает и какие задачи он может выполнять. Давайте разберемся!

1. AsyncTask был разработан для выполнения недолгих (в несколько секунд) операций;
2. Требует запуска в главном потоке;
3. Чаще всего его создают и запускают прямо в Activity.

Рассмотрим интерфейс класса. Прежде всего, как и в случае с другими классами фреймворка Android, нам нужно создать класс-наследник AsyncTask. Далее, что бросается в глаза – класс требует generic-параметры. Их три и они означают следующее: тип входного аргумента, тип значения прогресса, тип возвращаемого значения.

Далее нам нужно реализовать методы AsyncTask. Обязательным является метод **doInBackground**(), который получает на вход переменное количество параметров первого типа и возвращает третий тип. Как следует из названия, код этого метода выполняется в фоновом потоке. Однако до начала выполнения кода в doInBackground() выполнится код в методе **onPreExecute**(), причем выполнится он в главном потоке, что означает, что мы имеем доступ к интерфейсу. Например, мы можем показать progress bar до выполнения фоновой операции.

После выполнения операции в doInBackground(), результат выполнения фоновой операции попадает на вход **onPostExecute**(), который также выполняется в главном потоке, то есть мы можем отобразить результат в интерфейсе.

Что качается отображения прогресса, то в методе **doInBackground**() можно вызвать метод **publishProgress**() и передавать ему на вход аргумент второго типа. Это приведет к тому, что вызовется метод **onProgressUpdate**(), который сработает тоже в главном потоке.

Также запущенную операцию можно отменить. Для этого на AsyncTask вызывается метод **Cancel**(), принимающий на вход булевые значения: true, если мы разрешаем системе самой убить процесс. Альтернативно, мы сами можем проверять значения задачи внутри doInBackground() с помощью метода **isCanceled**(). Обработка отмененной операции будет происходить внутри метода **onCanceled**(), который также сработает в главном потоке.

```
mSampleTask.execute(10L) //запуск задачи
mSampleTask.cancel(true) //отмена задачи

class SampleTask extends AsyncTask<Long, Integer, String> {
    void onPreExecute();    //UI поток, перед фоновой операцией
    String doInBackground(Long...longs);    //фоновый поток

    void onPostExecute(String s);    //UI поток, после фоновой операции
    void publishProgress(1,2,3,4);  //внутри фоновой операции

    void onProgressUpdate(Integer...values);    //в UI потоке

    boolean isCanceled();   //внутри фоновой операции
    void onCanceled();  //в UI потоке

}
```

Давайте разберем пример. Создадим AsyncTask, который будет загружать картинку из интернета. На вход будет подан URL картинки. На выходе будет BitMap. Создадим разметку, на которой определим ImageView, Button и круглую ProgressBar. ProgressBar сделаем изначально невидимой (visibility: invisible). Изначально у нас пустой экран с кнопкой, при нажатии на которую появлется ProgressBar. После окончания загрузки ProgressBar исчезает, а в ImageView появляется картинка.

```
private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {
    @Override
    protected void onPreExecute(){
        mProgressBar.setVisibility(View.VISIBLE);
    }

    @Override
    protected Bitmap doInBackground (String...strings){
       return getBitmap(strings[0]);
    }

    @Override
    protected void onPostExecute(Bitmap bitmap) {
        mImageView.setImageBitmap(bitmap);
        mProgressBar.setVisibility(View.INVISIBLE);
```

```
    }
}
```

Код AsyncTask расположен в теле Activity и является внутренним классом. Входной параметр – строка, так как AsyncTask принимает на вход URL картинки. Параметр прогресса – void, мы не будем показывать прогресс процесса. Выходной параметр – Bitmap. Процесс показа и скрытия Progress Bar очевиден и прост. Код скачивания картинки выглядит вот так. Это не лучшее решение, но в качестве демонстрационного варианта самое то. В конце раздела мы покажем лучший вариант.

```
private Bitmap getBitmap(String url) {
    try {
        InputStream is = (InputStream) new URL(url).getContent();
        Bitmap d = BitmapFactory.decodeStream(is);
        is.close();
        return d;
    } catch (Exception e) {
        return null;
    }
}
```

Далее на кнопку вешаем Listener, на нем создаем и вызываем AsyncTask. В метод Execute передаем URL картинки.

```
mProgressBar = findViewById(R.id.progressBar);
mImageBar = findViewById(R.id.image);

Button startBtn = (Button) findViewById(R.id.btn_start);
startBtn.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        String url =
         ↪  "http://i0.kym-cdn.com/entries/icons/mobile/000/013/564/doge.jpg";

        new DownloadImageTask.execute(url);
      }
});
```

Выглядит все просто, но как бы не так! Ссылаясь внутри AsyncTask на элемент Activity, мы удерживаем контекст. Если во время работы AsyncTask мы перевернем телефон, тем самым вызвав пересоздание Activity, то уничтоженное Activity не сможет быть собрано сборщиком мусора до тех пор, пока AsyncTask продолжает свою работу. То есть у нас возникает проблема утечки контекста и решать ее можно по-разному, но суть одна – избавиться от жесткой ссылки на активити.

```
@Override
    protected void onPostExecute(Bitmap bitmap) {
        mImageView.setImageBitmap(bitmap);  //ссылка на поле активити
        mProgressBar.setVisibility(View.INVISIBLE);  //тоже
    }
}
```

Рассмотрим преобразование активити в статичный внутренний класс. Однако теперь мы не можем ссылаться на поля активити. Что же делать? Мы можем передать активити в конструкторе AsyncTask, а внутри самого AsyncTask хранить слабую ссылку на активити. Обратите внимание на конструктор: он принимает активити и сохраняет его не просто в поле, а в слабую ссылку. Далее каждый раз, когда нам нужно обратиться к полям активити, мы сначала достаем его из WeakReference и проверяем его на не null, таким образом не мешаем сборщику мусора утилизировать активити.

```
private static class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {
    private WeakReference<MainActivity> mActivityWeakReference;

    private DownloadImageTask(MainActivity activity) {
        mActivityWeakReference = new WeakReference<>(activity);
    }

    ...

    @Override
    protected void onPostExecute(Bitmap bitmap) {
        MainActivity activity = mActivityWeakReference.get();
        if (activity != null) {
            activity.getImageView().setImageBitmap(bitmap);
            activity.getProgressBar().setVisibility(View.INVISIBLE);
        }
    }
```

```
18  }
```

И посмотрим теперь, как изменяется вызов. Как было сказано ранее, теперь мы в конструктор AsyncTask передаем ссылку на наше активити. Также мы добавляем простые get*() для доступа к View-элементам. В нашем случае их больше не надо хранить как поля.

```
1   new DownloadImageTask(MainActivity.this).execute(url);
2
3   ...
4
5   ImageView getImageView() {
6       return findViewById(R.id.image);
7   }
8
9   ProgressBar getProgressBar() {
10      return findViewById(R.id.progressBar);
11  }
```

AsyncTask дает нам простые и понятные методы. Мы можем явно указать, что делать перед фоновой операцией, в фоне, что делать с результатом. Мы можем передать прогресс операции и что делать в случае отмены. Настройка через generic-параметры непривычна, но вполне понятна. Для недолгих простых задач это вполне хороший вариант.

Однако нельзя забывать про необходимость обработки утечки контекста, что усложняет использование AsyncTask. Также минус состоит в том, что AsyncTask не подходит для решения долгих задач, ведь пользователю придется ждать завершения окончания операции, не покидая активити. Также нужно отметить, что AsyncTask нужно запускать только в UI-потоке.

Бонус: скачивание картинки из интернета. Добавьте библиотеку Picasso в ваш проект через строку implementation и вызываете следующий код:

```
1   dependencies {
2       implementation "com.squareup.picasso:picasso:2.3.2"
3       ...
4   }
5   Picasso.with(this).load(url).into(mImageView);
```

Picasso – очень удобная библиотека для работы с изображениями. В метод with() передаем кон-

текст, далее по цепочке в метод load() передаем URL изображения, далее передаем в метод into() контейнер. И это минимальные необходимые действия, чтобы скачать изображение.

Полный код урока:

```
public class MainActivity extends AppCompatActivity {
    private DownloadImageTask mDownloadImageTask;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        final String url =
         ↪  "http://i0.kym-cdn.com/entries/icons/mobile/000/013/564/doge.jpg";

        mDownloadImageTask = new DownloadImageTask(MainActivity.this);

        Button startBtn = findViewById(R.id.btn_start);
        startBtn.setOnClickListener(new View.OnClickListener() {

            @Override
            public void onClick(View v) {
                mDownloadImageTask.execute(url);
            }

        });

        Button btnCancel = findViewById(R.id.btn_cancel);
        btnCancel.setOnClickListener(new View.OnClickListener() {

            @Override
            public void onClick(View v) {
                mDownloadImageTask.cancel(true);
            }
        });
    }

    ImageView getImageView() {
        return findViewById(R.id.imageView);
    }
```

```
35
36      ProgressBar getProgressBar() {
37          return findViewById(R.id.progressBar);
38      }
39
40      private static class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {
41          private WeakReference<MainActivity> mActivityWeakReference;
42          private DownloadImageTask(MainActivity activity) {
43              mActivityWeakReference = new WeakReference<>(activity);
44          }
45
46          @Override
47          protected void onPreExecute() {
48              MainActivity activity = mActivityWeakReference.get();
49              if (activity != null)
50                  activity.getProgressBar().setVisibility(View.VISIBLE);
51          }
52
53          @Override
54          protected Bitmap doInBackground(String... strings) {
55              return getBitmap(strings[0]);
56          }
57
58          private Bitmap getBitmap(String url) {
59              try {
60                  TimeUnit.SECONDS.sleep(5);
61                  if (isCancelled()) {
62                      return null;
63                  }
64
65                  InputStream is = (InputStream) new URL(url).getContent();
66                  Bitmap d = BitmapFactory.decodeStream(is);
67                  is.close();
68                  return d;
69              } catch (Exception e) {
70                  return null;
71              }
72          }
73
74          @Override
```

```
        protected void onPostExecute(Bitmap bitmap) {
            MainActivity activity = mActivityWeakReference.get();
            if (activity != null) {
                activity.getImageView().setImageBitmap(bitmap);
                activity.getProgressBar().setVisibility(View.INVISIBLE);
            }
        }

        @Override
        protected void onCancelled() {
            MainActivity mainActivity = mActivityWeakReference.get();
            if (mainActivity != null) {
                mainActivity.getProgressBar().setVisibility(View.INVISIBLE);
                Toast.makeText(mainActivity, "Canceled", Toast.LENGTH_SHORT).show();
            }
        }
    }
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
                xmlns:app="http://schemas.android.com/apk/res-auto"
                xmlns:tools="http://schemas.android.com/tools"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:padding="16dp">

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true" />

    <Button
        android:id="@+id/btn_start"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_above="@id/btn_cancel"
```

```
20          android:text="Start" />
21
22      <Button
23          android:id="@+id/btn_cancel"
24          android:layout_width="match_parent"
25          android:layout_height="wrap_content"
26          android:layout_alignParentBottom="true"
27          android:text="Cancel" />
28
29      <ProgressBar
30          android:id="@+id/progressBar"
31          android:layout_width="wrap_content"
32          android:layout_height="wrap_content"
33          android:layout_centerInParent="true"
34          android:visibility="invisible" />
35
36  </RelativeLayout>
```

### 1.3.2. HaMeR – Handler-Message-Runnable

В этом разделе мы рассмотрим пример межпотокового взаимодействия при помощи паттерна HaMeR. Сразу же перейдем к примеру. Приложение бессмысленное и беспощадное, но при этом забавное. В ImageView у нас находится картинка. При нажатии на кнопку картинка передается в фоновый поток, в котором она разбирается на пиксели, в каждом из которых происходит смещение rgb-канала: значение из g записывается в r, значение из r записывается в b, значение из b записывается в g. Пока идет процесс смещения, на интерфейсе ползет Progress Bar. После завершения смещения картинка передается в интерфейс. На этом все. Теперь давайте разберемся в том, что под капотом.

Начнем издалека. Как вообще запускаются приложения в Android? Если на текущий момент на устройстве нет запущенных компонентов нашего приложения, то для этого приложения система выделяет отдельный процесс, в котором оно будет работать. Каждое приложение в Android работает в своем собственном процессе, в своей области памяти. В память чужого процесса попасть нельзя, это гарантируется системой.

Первый поток, запущенный в процессе, называется **главным потоком** – main thread – отвечает

еще и за рисование интерфейса (UI thread). Все операции с интерфейсом происходят в главном потоке. Из других потоков изменять UI нельзя.

Как обычно работают потоки в Java? Поток выполняет свои инструкции, при необходимости записывает куда-то результат и благополучно самоуничтожается. Поток в представленном ниже коде выведет 15 и самоуничтожится.

```
private class MyThread extends Thread {
    @Override
    public void run() {
        super.run();

        int a = 5;
        int b = 10;
        int c = a + b;
        System.out.println(c);
    }
}
MyThread thread = new MyThread();
thread.start();
```

Создание потока – дорогая с точки зрения ресурсов операция. И пусть ThreadPoolExecutor'ы и решают эту задачу, речь сейчас не о них. Android-приложения работают несколько иначе. После выполнения кода в onCreate(), onStart() и onResume(), наше приложение не завершается, поток не уничтожается и, по-видимому, ничего не происходит, до тех пор, пока наш пользователь не нажмет что-нибудь. И тогда приложение отреагирует на действия пользователя должным образом: откроет новый экран, поменяет интерфейс или еще что-нибудь произойдет. Затем приложение снова начнет ждать пользователя, причем ждать оно может вплоть до окончания заряда батарейки. В связи с этим возникаем закономерный вопрос: почему поток MainThread не умирает? Он не умирает потому, что проделываем бесконечную работу: с MainThread ассоциирован объект типа Looper, который находится в бесконечном цикле, не давая MainThread уничтожиться и проверяя, не произошло каких-либо изменений со стороны пользователя, которые могли бы потребовать реагирования.

Откуда Looper узнает про действия пользователя? С Looper ассоциирована очередь сообщений MessageQueue. Сообщения – объекты класса Message – являют собой действия или команды. Looper проверяет MessageQueue: если в неи есть новое сообщение, то он извлекает его и отправляет на обработку.

Как сообщения попадают в MessageQueue и куда Looper отправляет их на обработку? Существует еще один специальный объект типа Handler, который связан с Looper. У Handler может быть только один Looper, но у Looper'а может быть сколь угодно Handler'ов. Handler отправляет сообщения в очередь своего Looper'а. У сообщения есть ссылка на Handler, который это сообщение должен обработать. Системные сообщения обрабатываются системными Handler'ами.

Аббревиатура HaMeR ссылается на Handler, Message и Runnable. Казалось бы, причем тут Runnable? Дело в том, что Handler может отправлять в очередь не только сообщения, но и Runnable-объекты. Причем код внутри метода run() будет выполняться в том потоке, к Looper'у которого привязан Handler, который это сообщение отправил.

Кратко пройдемся по тому, как устроен MainThread. В MainThread'е находится объект Looper. У Looper'а в бесконечном цикле проверяют MessageQueue. Если в ней находится новый Message, его отправляют на обработку. Передачу и обработку системных сообщений, наподобие нажатия кнопки или swipe, Android берет на себя. Если же мы ходим добавить обработку своих сообщений, нам нужно создать объект типа Handler. Handler умеет отправлять сообщения в очередь MessageQueue. В Message есть ссылка на Handler, который должен его обработать. Это может быть как тот Handler, который его отправил, так и какой-то другой Handler.

Теперь рассмотрим, как **создать Handler**. Когда мы создаем Handler с помощью конструктора без параметров, он привязывается к Looper'у текущего потока. Мы можем передать в конструктор Looper и тогда Handler привяжется к нему. И очень просто можно привязать Handler к Looper'у главного потока при помощи статического метода getMailLooper(). Для отправки Runnable используется метод post() и различные его вариации. Runnable затем оборачивается в Message и попадает в MessageQueue. Looper достает его и отправляет обратно тому же Handler'у, на обработку. Просто post() в этом случае не очень полезен, однако мы можем воспользоваться методом postDelayed, которому помимо Runnable передаем еще и задержку в миллисекундах, после которой выполнится код в run().

```
1  Handler handler = new Handler(); //привязывается к looper'у текущего потока
2  Handler bgHandler = new Handler(bgLooper); //привязывается к переданному Looper'у
3  Handler mainHandler = new Handler(Looper.getMailLooper()); //привязывается к Looper'у
   ↪  главного потока
4
5  handler.post(new Runnable() {
6      @Override
7      public void run() {
8          // этот код выполнится в том потоке, к которому привязан handler
9      }
```

```
});
handler.postDelayed(new Runnable(), 4000); //код выполнится через 4000мс
```

Посмотрим на отправку обычного сообщения. Объекты типа Message не надо создавать самому. Лучше воспользоваться одним из статических методов, которые возвращают сообщение из общего пула сообщений. Мы тем самым переиспользуем уже созданное сообщение. Если мы хотим, чтобы сообщение обработалось тем же Handler'ом, который его отправляет, нужно воспользоваться методом obtainMessage(). У этого метода много перегруженных вариантов. Вариант на слайде принимает Id сообщения (поле WHAT, и данные в виде сообщения (поле Object). Далее мы можем через точку добавить метод sendToTarget и тем самым отправить сообщение на обработку в MessageQueue.

```
mHandler.obtainMessage(MESSAGE_WHAT, dataObject).sendToTarget();
```

Теперь давайте разберем обработку сообщения. При создании Handler'а можно переопределить его метод handleMessage(). В нем мы проверяем поле what сообщения и, в нашем случае, извлекаем данные и отправляем их в метод. В конце вызываем метод recycle(), который возвращает сообщение в пул и делает его готовым для переиспользования. Отправка и обработка такого сообщения практически идентична вызову post с методом внутри.

```
mHandler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case MESSAGE_WHAT: {
                DataObject dataObject = (DataObject) msg.obj;
                doSomething(DataObject);

                msg.recycle();
            }
        }
    }

};
mHandler.post(new Runnable() {
    @Override
    public void run() {
        doSomething(DataObject);
```

```
19      }
20  }
```

Если же мы хотим отправить сообщение в другой Handler, то мы можем воспользоваться статическим методом obtain. Он вызывается из класса Message и тоже имеет много перегруженных версий. На слайде мы указываем Handler-адресат, Id сообщения и данные и затем с помощью доступного нам Handler'а отправляем сообщение. Найдя это сообщение в очереди, Looper отправит его нужному Handler'у.

```
1  Message message = Message.obtain(mAnotherHandler, MESSAGE_ANOTHER_WHAT,
    ↪  ANOTHER_OBJECT);
2
3  mHandler.sendMessage(message);
```

HandlerThread – это класс, который наследуется от обычного JavaThread и имеет внутри себя настроенный Looper и MessageQueue. Как это обычно происходит в Android, мы можем отнаследоваться от базового класса и добавить свою логику. После создания HandlerThread нам нужно запустить его с помощью метода Start. Затем нужно вызвать метод getLooper, который форсирует создание и запуск Handler'а. GetLooper() приводит к тому, что срабатывает CallbackHandlerReda onLooperPrepared(). Этот метод можно переопределить и провести в нем создание наших Handler'ов. Мы создаем Handler'ы, которые привязаны к MainThread'у, то есть получаем связь фонового потока с главным посредством Handler'а.

```
1  MyHandlerThread extends HandlerThread
2
3  mMyHandlerThread.start();
4  mMyHandlerThread.getLooper();
5
6  @Override
7  protected void onLooperPrepared() {
8      mMainHandler = new Handler(Looper.getMainLooper());
9  }
```

Вообще, в этом и заключается суть подхода HaMeR: связь главного и фонового потока через Handler'ы, отправка сообщений и Runnable из потока в поток и обработка в нужных местах.

Полный код приложения. Итак, у нас есть два класса: первый – Activity, второй – реализация HandlerThread. Код основательно покрыт комментариями, все должно быть понятно.

**MainActivity.java**

```
public class MainActivity extends AppCompatActivity implements
 ↪  ImageProcessThread.Callback{

  //весь код ниже выполняется в мейнтреде
  private ImageView mDoge;
  private ProgressBar mProgressBar;
  private ImageProcessThread mImageProcessThread;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      final Button performBtn = findViewById(R.id.btn_perform);
      mDoge = findViewById(R.id.iv_doge);
      mProgressBar = findViewById(R.id.progress);
      //создаем новый экземпляр фонового потока потока
      mImageProcessThread = new ImageProcessThread("Background ");

      //запускаем поток и инициализируем Looper
      mImageProcessThread.start();
      mImageProcessThread.getLooper();  // -> вызовется onLooperPrepared()
      mImageProcessThread.setCallback(this);
      //теперь в фоновом потоке будет крутиться лупер и ждать задач

      performBtn.setOnClickListener(new View.OnClickListener() {
          @Override
          public void onClick(View v) {
              //выдергиваем Bitmap из ImageView и скармливаем в наш поток,
              //просто вызывая метод на нем
              BitmapDrawable drawable = (BitmapDrawable) mDoge.getDrawable();
              mImageProcessThread.performOperation(drawable.getBitmap());
          }
      });
  }

```

```
35      // методы колбека из интерфейса
36      @Override
37      public void sendProgress(int progress) {
38          mProgressBar.setProgress(progress);
39      }
40
41      @Override
42      public void onCompleted(Bitmap bitmap) {
43          mDoge.setImageBitmap(bitmap);
44      }
45
46      @Override
47      protected void onDestroy() {
48          //гасим фоновый поток, не мусорим
49          mImageProcessThread.quit();
50          super.onDestroy();
51      }
52  }
```

### activity_main.xml

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3                 xmlns:tools="http://schemas.android.com/tools"
4                 android:layout_width="match_parent"
5                 android:layout_height="match_parent"
6                 android:gravity="center"
7                 android:orientation="vertical"
8                 tools:context="com.e_legion.coursera.MainActivity">
9
10    <ProgressBar
11        android:id="@+id/progress"
12        style="@style/Widget.AppCompat.ProgressBar.Horizontal"
13        android:layout_width="match_parent"
14        android:layout_height="wrap_content"
15        android:layout_marginBottom="16dp"
16        android:indeterminate="false" />
17
18    <ImageView
```

```
        android:id="@+id/iv_doge"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/doge" />

    <Button
        android:id="@+id/btn_perform"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="ВЖУХ!" />
</LinearLayout>
```

### ImageProcessThread.java

```
public class ImageProcessThread extends HandlerThread {
    private static final int MESSAGE_CONVERT = 0;
    private static final int PERCENT = 100;
    private static final int PARTS_COUNT = 50;
    private static final int PART_SIZE = PERCENT / PARTS_COUNT;

    private Handler mMainHandler;
    private Handler mBackgroundHandler;
    private Callback mCallback;

    public ImageProcessThread(String name) {
        super(name);
    }

    public void setCallback(Callback callback) {
        mCallback = callback;
    }

    @SuppressLint("HandlerLeak")  // находимся в хендлер треде, утечки не будет
    @Override
    protected void onLooperPrepared() {
        //создаем хендлер, связанный с мейнтредом
        mMainHandler = new Handler(Looper.getMainLooper());

        //также создаем хендлер, связанный с текущим тредом
```

```
26          mBackgroundHandler = new Handler() {
27              //и указываем ему, что делать в случае получения сообщения с нашим what
                ↪  значением
28              // это будет выполнено в фоновом потоке
29              @Override
30              public void handleMessage(Message msg) {
31                  switch (msg.what) {
32                      case MESSAGE_CONVERT: {
33                          //выдергиваем битмапу и процессим ее
34                          Bitmap bitmap = (Bitmap) msg.obj;
35                          processBitmap(bitmap);
36                msg.recycle();
37                      }
38                  }
39              }
40          };
41      }
42
43      private void processBitmap(final Bitmap bitmap) {
44          //бессмысленно-беспощадная долгая операция
45          //здесь я выдергиваю пиксели из битмапы
46          int h = bitmap.getHeight();
47          int w = bitmap.getWidth();
48          int[] pixels = new int[h * w];
49          bitmap.getPixels(pixels, 0, w, 0, 0, w, h);
50
51          // java - лайфхак - костыль - использую final массив с одной ячейкой, в которой
                ↪  буду менять процент.
52          // в функциональные интерфейсы мы можем передавать только финальные или
                ↪  эффективно финальные значения
53          // но процент у меня меняется, и я обхожу это требование
54          final int[] progress = new int[1];
55
56          //потом прохожусь по каждому пикселю и сдвигаю хекс значения
57          //красный на зеленый, зеленый на синий, синий на красный
58          for (int i = 0; i < h * w; i++) {
59              String hex = String.format("#%06X", (0xFFFFFF & pixels[i]));
60              String R = hex.substring(1, 3);
61              String G = hex.substring(3, 5);
62              String B = hex.substring(5);
```

```
63              String mess = B + R + G;
64              pixels[i] = Integer.parseInt(mess, 16);
65
66              //здесь логика показа процента готовности обработки изображения
67              //показываем PARTS_COUNT (50) кусочков в прогресс баре
68              int part = w * h / PARTS_COUNT;
69
70              if (i % part == 0) {
71                  progress[0] = i / part * PART_SIZE; // <- костыльная магия
72                  //постим в мейнтред через мейнхендлер
73                  mMainHandler.post(new Runnable() {
74                      @Override
75                      public void run() {
76                          // этот код уже выполняется в главном потоке
77                          mCallback.sendProgress(progress[0]);
78                      }
79                  });
80              }
81          }
82
83          //создаем битмапу из пикселей с уже смещенным цветом
84          final Bitmap result = Bitmap.createBitmap(pixels, w, h, Bitmap.Config.RGB_565);
85          //постим в мейнтред
86          mMainHandler.post(new Runnable() {
87              @Override
88              public void run() {
89                  //этот код так же исполнится в мейнтреде
90                  mCallback.onCompleted(result);
91              }
92          });
93      }
94
95      //этот метод будет вызываться  из главного потока
96      public void performOperation(Bitmap inputData) {
97          // создаем Message от BackgroundHandler'а, записываем в него Bitmap и отправляем
            ↪  в очередь
98          mBackgroundHandler
99                  .obtainMessage(MESSAGE_CONVERT, inputData)
100                 .sendToTarget();
101         // созданное сообщение попадает в очередь фонового потока,
```

```
102         // так как хендлер связан с лупером фонового потока
103         // сейчас вызовется метод handleMessage(),
104         // который переопределен выше, в методе onLooperPrepared
105     }
106
107     public interface Callback {
108         void sendProgress(int progress);
109         void onCompleted(Bitmap bitmap);
110     }
111 }
```

### 1.3.3. Loader, знакомство

В этом разделе мы рассмотрим еще один механизм асинхронной загрузки – Loader'ы. Они позволяют загружать данные в фоновом потоке, не привязываясь к активити или к фрагменту, с которого они были запущены. В этом их преимущество: мы можем запустить их в одном активити, на одном экране, а получить результаты и обработать их в другом активити, на другом экране. Сейчас давайте сделаем шаг назад и разберем классы, которые входят в API.

Для запуска Loader нам нужно получить экземпляр LoaderManager. **LoaderManager** – класс, который запускает Loader. Для получения экземпляра менеджера – LoaderManager.getSupportLoaderManager(). Есть еще метод **getLoaderManager**(), но по факту onCompatActivity и Support-фрагменте возвращают одно и то же. Дальше, чтобы запустить Loader, нужно воспользоваться методом **InitLoader**(). На вход он принимает уникальный идентификатор Loader'a, Bundle с аргументами и ссылку на реализацию callback'ов. Если система определяет, что Loader'a с таким id не существует, то она его создаст. Если он уже существует, то она его переиспользует. Generic означает тип данных. Заметим, что у Loader'a и у Callback'ов тип данных одинаковый. Bundle – это необязательные аргументы, передаваемые в Loader и используемые во время первого создания. LoaderCallbacks – callback-интерфейс с методами для передачи состояния Loader'a вызывающей стороне. Если нам нужно перезапустить текущую загрузку с новыми аргументами (данными), можно воспользоваться методом **restartLoader**(). Он также создаст Loader, если запущенного Loader'a с таким id не существует. Из других полезных методов есть **hasRunningLoaders**(), возвращающий true, если Loader запущен и еще не вернул значение возвращающей стороне. **GetLoader**() принимает на вход id и возвращает Loader, если такой Loader существует, или null, если он еще не был создан. **DestroyLoader** останавливает загрузку, и уничтожает Loader с переданным id.

```
Loader<D> initLoader(int id, Bundle args, LoaderManager.LoaderCallbacks<D> callback)
Loader<D> restartLoader(int id, Bundle args, LoaderManager.LoaderCallbacks<D>
    ↪ callback)

boolean hasRunningLoaders();
Loader<D> getLoader(int id);
void destroyLoader(int d);
```

Для обеспечения взаимодействия между системой и нашим кодом, мы реализуем интерфейс LoaderCallbacks и передаем реализацию метода инициализации Loader. Чаще всего реализация происходит в активити или во фрагменте, но это не обязательно. Всего нам нужно переопределить три метода:

1. **onCreateLoader(int, Bundle)**. Мы возвращаем в систему Loader, проверяя переданный id и используя Bundle, если нужно. Этот метод вызывается, если после выполнения InitLoader() и StartLoader(), такого метода еще нет.
2. **onLoadFinished(Loader<D>, D)**. Вызывается, когда загрузка данных завершена. Решаем, что делать с полученными данными (чаще всего показываем их в списке).
3. **onLoaderReset(Loader<D>)**. Вызывается, когда созданный Loader уничтожается через метод onDestroyLoader() или когда был уничтожен активити или фрагмент. Данные нм больше не нужны. Нам нужно убрать все ссылки для сборщика мусора.

Теперь, после знакомства с классами и компонентами, настала очередь поговорить про сами Loader'ы. Во-первых, **Loader** – базовый класс для всех Loader'ов. Их две: **AsyncTaskLoader** – обертка, которая инкапсулирует AsyncTask, решая некоторые ее проблемы, и **CursorLoader**, который использует курсор, объекты типа Cursor, для получения данных с бд или с контент-провайдера. По идее, этих двух лоадеров должно быть достаточно для решения большинства задач, но если вдруг вы хотите реализовать что-то действительно уникальное, то просто наследуйтесь от одной из реализаций или даже базового класса. Пример работы с Loader мы рассмотрим чуть дальше по курсу, когда будем изучать списки.

### 1.3.4. ContentProvider, знакомство

В этом видео мы разберем, что такое ContentProvider.

**ContentProvider** – механизм, который инкапсулирует доступ к данным, дополнительно позволяя обращаться к ним из разных процессов. Если говорить в двух словах, то это интерфейс – публичные элементы и методы, которые может вызывать третья сторона – для доступа к данным. Данные при этом могут храниться как угодно: в SQL-базе данных, Preference'ах и т.д. Первый вопрос, который может возникнуть: зачем нас вообще оборачивать доступ к нашим данным в какой-то интерфейс? Ответ: дело в том, что ContentProvider позволяет делиться нашими данными с другими процессами и приложениями. И дают этим приложениям возможность безопасно для нашего приложения манипулировать нашими данными. При этом, так как стороннее приложение обращается к нашим данным через интерфейс ContentProvider, мы можем в одном из обновлений нашего приложения можем просто перенести данные из SQL-базы в любую другую, также переделав реализацию ContentProvider, а вызывающая сторона вообще не заметит никакой разницы: она будет работать так же, как и раньше.

Естественно, такой мощный механизм не мог обойтись без системной реализации. Христоматийный пример использования ContentProvider – доступ к телефонной книге из нашего приложения. Различные мессенджеры используют телефонную книгу, чтобы считывать контакты для отправки сообщений и добавлять новые контакты. Банковские приложения, например, проверяют возможность передачи денег по номеру телефона. Для этого нужно, чтобы ваш контакт был зарегистрирован в том же банке и с тем же номером телефона, а дальше дело техники и простого сравнения.

Другой пример – мультимедиа файлы. Изображения, видео, музыка – ко всем медиафайлам можно получить доступ через ContentProvider. Когда вы запускаете плеер, он ищет mp3-файлы при помощи ContentProvider. Хотите переслать картинки другу в чате? Поиск происходит через ContentProvider. Выбираете аватарку для игры – камера или, опять же, ContentProvider.

Чуть более абстрактный пример – добавление события в календарь. С Google-аккаунтом в телефоне связан календарь, в который можно добавлять события. Всякого рода приложения-органайзеры могут считывать события из этого календаря и, соответственно, добавлять туда что-то свое. Также можно добавлять их через gmail, отправляя приглашения просто по почте.

Сейчас мы хотели бы сделать некоторое уточнение. Для работы с ContentProvider используется CursorLoader, который вскользь упоминался в предыдущем разделе. Для того, чтобы во всей красе насладиться работой ContentProvider, нам нужно научиться выводить на экран списки. Поэтому в следующем уроке мы научимся создавать экраны со списками и помимо прочего, разберем, как получить данные для заполнения списка с помощью ContentProvider.

### 1.3.5. Материалы для самостоятельного изучения

HaMeR:

1. https://code.tutsplus.com/tutorials/concurrency-on-android-using-hamer-framework–cms-27129
2. https://code.tutsplus.com/tutorials/practical-concurrency-on-android-with-hamer–cms-27137

Loaders:

1. https://developer.android.com/guide/components/loaders.html?hl=
2. http://androiddocs.ru/loaders-ispolzuem-asynctaskloader
3. https://medium.com/@sanjeevy133/an-idiots-guide-to-android-asynctaskloader-76f8bfb0a0c0

ContentProvider и Cursor:

1. https://developer.android.com/guide/topics/providers/content-providers.html?hl=ru
2. https://youtu.be/zeDzbzLmpLs
3. http://developer.alexanderklimov.ru/android/sqlite/cursor.php

**О проекте**
Академия e-Legion – это образовательная платформа для повышения квалификации в мобильной разработке. Слушайте лекции топовых разработчиков, выполняйте практические задания и прокачивайте свои скиллы. Получите высокооплачиваемую профессию – разработчик мобильных приложений.

## Программа “Многопоточность и сетевое взаимодействие”

### Блок 1. Обзор средств для обеспечения многопоточности

- Знакомство с курсом
- Многопоточность и параллельное программирование
- Обзор инструментов для обеспечения многопоточности в Java (Thread, Runnable, Callable, Future, Executors)
- Обзор инструментов для обеспечения многопоточности в Android (IntentService + BroadcastReceiver, HaMeR, AsyncTask, Loaders)
- Материалы для самостоятельного изучения
- Задание на программирование. Многопоточность.

### Блок 2. Service + BroadcastReceiver

- Знакомство с Service, IntentService
- Создание Service
- Бродкастресивер, знакомство
- Создание BroadcastReceiver
- Связка Activity-Service-BroadcastReceiver-Activity
- Материалы для самостоятельного изучения
- Задание на программирование. Многопоточность.

### Блок 3. Многопоточность в Android

- AsyncTask, знакомство
- AsyncTask, работа
- HaMeR
- Пример работы HaMeR
- Loader, знакомство

- ContentProvider, знакомство
- Материалы для самостоятельного изучения
- Задание на программирование. Многопоточность.