



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Практическое занятие № 4/4ч.

Разработка мобильных компонент анализа безопасности информационно-аналитических систем

	<i>(наименование дисциплины (модуля) в соответствии с учебным планом)</i>
Уровень	бакалавриат
	<i>(бакалавриат, магистратура, специалитет)</i>
Форма обучения	очная
	<i>(очная, очно-заочная, заочная)</i>
Направление(-я) подготовки	10.05.04 Информационно-аналитические системы безопасности
	<i>(код(-ы) и наименование(-я))</i>
Институт	комплексной безопасности и специального приборостроения ИКБСП
	<i>(полное и краткое наименование)</i>
Кафедра	КБ-3
	<i>(полное и краткое наименование кафедры, реализующей дисциплину (модуль))</i>
Используются в данной редакции с учебного года	2021/22
	<i>(учебный год цифрами)</i>
Проверено и согласовано « ____ » ____ 20 ____ г.	
	<i>(подпись директора Института/Филиала с расшифровкой)</i>

Москва 2021 г.

ОГЛАВЛЕНИЕ

1	Многопоточность.	3
1.1	Архитектура Android	3
1.2	Процессы.....	5
1.3	Жизненный цикл процессов.....	8
2	Потоки	10
2.1	Задание. Thread и Runnable	11
3	Передача данных между потоками	16
3.1	Задание. Thread в UI.....	16
3.2	Задание. Looper	17
3.3	Задание. Loaders	19
4	Service	26
4.1	Создание Service.....	27
4.2	Жизненный цикл служб	29
4.3	Запуск сервиса и управление его перезагрузкой	30
5	WorkManager.....	32
5.1	Задание.....	33
6	Контрольное задание.....	34

1 МНОГОПОТОЧНОСТЬ.

Требуется создать новый проект *ru.mirea.«фамилия».practice4*.

Многопоточность — это принцип построения программы, при котором несколько блоков могут выполняться одновременно. Потоки позволяют выполнять несколько задач одновременно, не мешая друг другу, что даёт возможность эффективно использовать системные ресурсы. Потоки используются в тех случаях, когда одно долгоиграющее действие не должно мешать другим действиям.

Например, есть музыкальный проигрыватель с кнопками воспроизведения и паузы. При нажатии на кнопку воспроизведения запускается музыкальный файл в отдельном потоке и не будет возможности нажать на кнопку паузы, пока файл не воспроизведётся полностью. С помощью многопоточности возможно обойти данное ограничение.

Для повышения отзывчивости приложения требуется переместить все медленные, трудоёмкие операции из главного потока приложения в дочерние.

Все компоненты приложения в Android, включая активности, сервисы и приёмники широковещательных намерений, начинают работу в главном потоке приложения. В результате трудоёмкие операции в любом из этих компонентов блокируют все остальные части приложения, включая сервисы и активности на переднем плане.

1.1 Архитектура Android

Центральными для системы являются демон *zygote* и запущенные им процессы (рисунок 1.1). Первый процесс, всегда запускаемый *zygote*, называется *system_server* и содержит все основные службы операционной системы. Основными частями являются диспетчер электропитания, диспетчер пакетов, оконный диспетчер и диспетчер активностей.

По мере необходимости из *zygote* создаются другие процессы. Некоторые из них станут постоянными процессами, являющимися частью основной операционной системы, например, телефонный стек в процессе телефона, который должен всегда

оставаться в работе. В ходе работы системы по мере необходимости будут создаваться и останавливаться дополнительные процессы приложений.

Взаимодействие приложений с операционной системой осуществляется за счет вызовов предоставляемых ею библиотек, совокупность и является средой Android (Android framework). Некоторые из библиотек могут работать в рамках данного процесса, но многим понадобится межпроцессорный обмен данными с другими процессами. Зачастую данный обмен ведется со службами в процессе `system_server`.

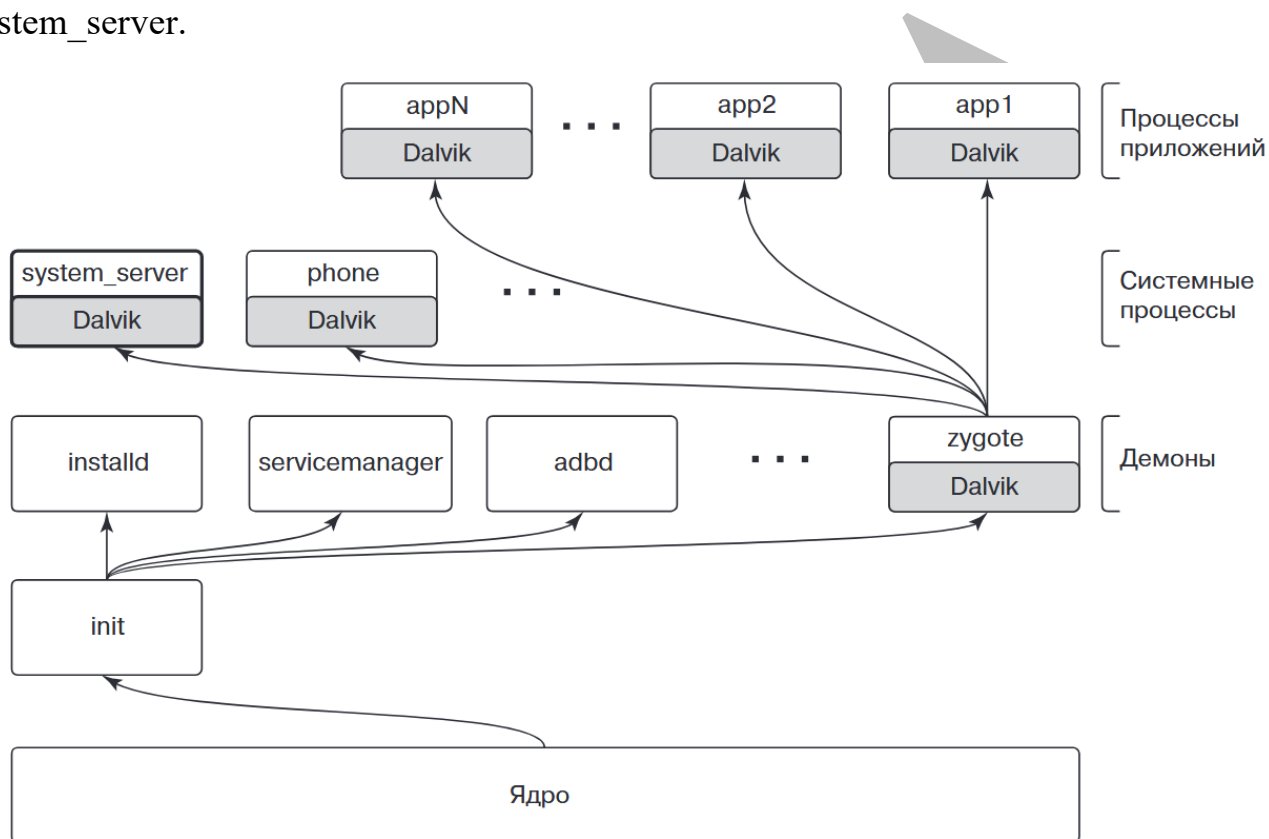


Рисунок 1.1 – Процесс загрузки ОС Android

В Linux каждая запущенная программа является отдельным процессом. Каждый процесс обладает уникальным номером и собственной «территорией» — виртуальным адресным пространством, в рамках которого содержатся все данные процесса.

Zygote отвечает за доставку инициализированной Dalvik-среды в точку, где готов запуск системного кода или кода приложения, написанного на языке Java. Все новые процессы, основанные на применении среды Dalvik (системные или прикладные), ответвляются от `zygote`, что позволяет им начинать выполнение с уже

готовой к работе средой. Zygote не только доставляет Dalvik, он также осуществляет предварительную загрузку многих частей Android-среды, которые обычно используются в системе и приложениях, а также загружает ресурсы и другие часто используемые компоненты.

1.2 Процессы

Для запуска нового процесса, диспетчер активностей должен быть связан с процессом zygote. При первом запуске диспетчера активностей создается выделенный сокет с zygote, через который посылает команду, когда нуждается в запуске нового процесса. Команда прежде всего дает описание создаваемой песочницы: UID, под которым должен запуститься новый процесс, и любые другие ограничения, связанные с мерами безопасности, которые будут применяться. Таким образом, zygote должен запускаться с root-правами: при разветвлении он выполняет соответствующую настройку для UID, с которым процесс будет запущен, и в конце сбрасывает root-права и изменяет процесс, присваивая ему нужный UID (таблица 1.1).

Таблица 1.1 – Значения UID в ОС Android

UID	Цель
0	Root
1000	Основная система (процесс system_server)
1001	Телефонные службы
1013	Медийные низкоуровневые процессы
2000	Доступ к оболочке командной строки
10000-19999	Динамически назначаемые UID-идентификаторы приложений
100000	Начало вторичных пользователей

При создании нового процесса из zygote используется Linux-функция fork, но вызов функции exec не применяется. Новый процесс является точной копией исходного процесса zygote, со всеми его инициализированными и уже установленными состояниями, и он сразу же готов к работе. Связь нового Java-процесса с исходным процессом zygote показана на рисунке 1.2. После ответвления

в распоряжении нового процесса оказывается собственная отдельная Dalvik-среда, таким образом, через страницы копирования при записи делит с *zygote* все заранее загруженные и инициализированные данные.

При наличии готового к работе нового процесса требуется установить правильную идентичность (UID и т. д.), завершить любые инициализации Dalvik-среды, которые требуются запускаемым потокам, и загрузить запускаемый код приложения или системный код.

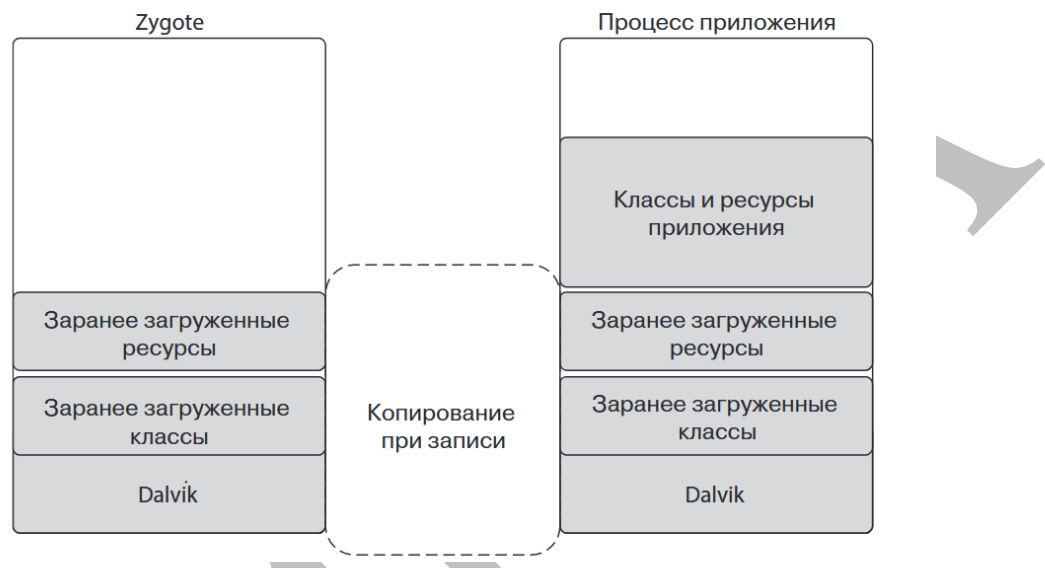


Рисунок 1.2 – Создание нового процесса

Порядок запуска активности в новом процессе показан на рисунке 1.3. Далее представлены подробности каждого этапа:

1. Один из существующих процессов (например, предназначенный для запуска приложений) осуществляет вызов диспетчера активностей с намерением, дающим описание новой активности, которую он собирается запустить.
2. Диспетчер активностей просит, чтобы диспетчер пакетов провел разрешение намерения до явного компонента.
3. Диспетчер активностей определяет, что прикладной процесс еще не запущен, а затем просит *zygote* создать новый процесс с соответствующим UID.
4. *Zygote* выполняет ветвление, создает новый процесс, являющийся клоном себя самого, сбрасывает права и устанавливает его UID песочнице приложения, а затем завершает инициализацию Dalvik в этом процессе для полноценной работы среды выполнения Java. Например, после ветвления должны запускаться такие

потоки, как сборщик мусора.

5. Новый процесс, представляющий собой клон `zygote` с полностью установленной и работающей Java-средой, осуществляет обратный вызов диспетчера активностей с вопросом: «Для чего я нужен?».

6. Диспетчер активностей возвращает ему полную информацию о запускаемом в нем приложении, например, о том, где найти его код.

7. Новый процесс загружает код запускаемого приложения.

8. Диспетчер активностей отправляет новому процессу любую ожидающую операцию, в данном случае «Запустить активность X».

9. Новый процесс получает команду на запуск активности, создает экземпляр соответствующего Java-класса и выполняет его.

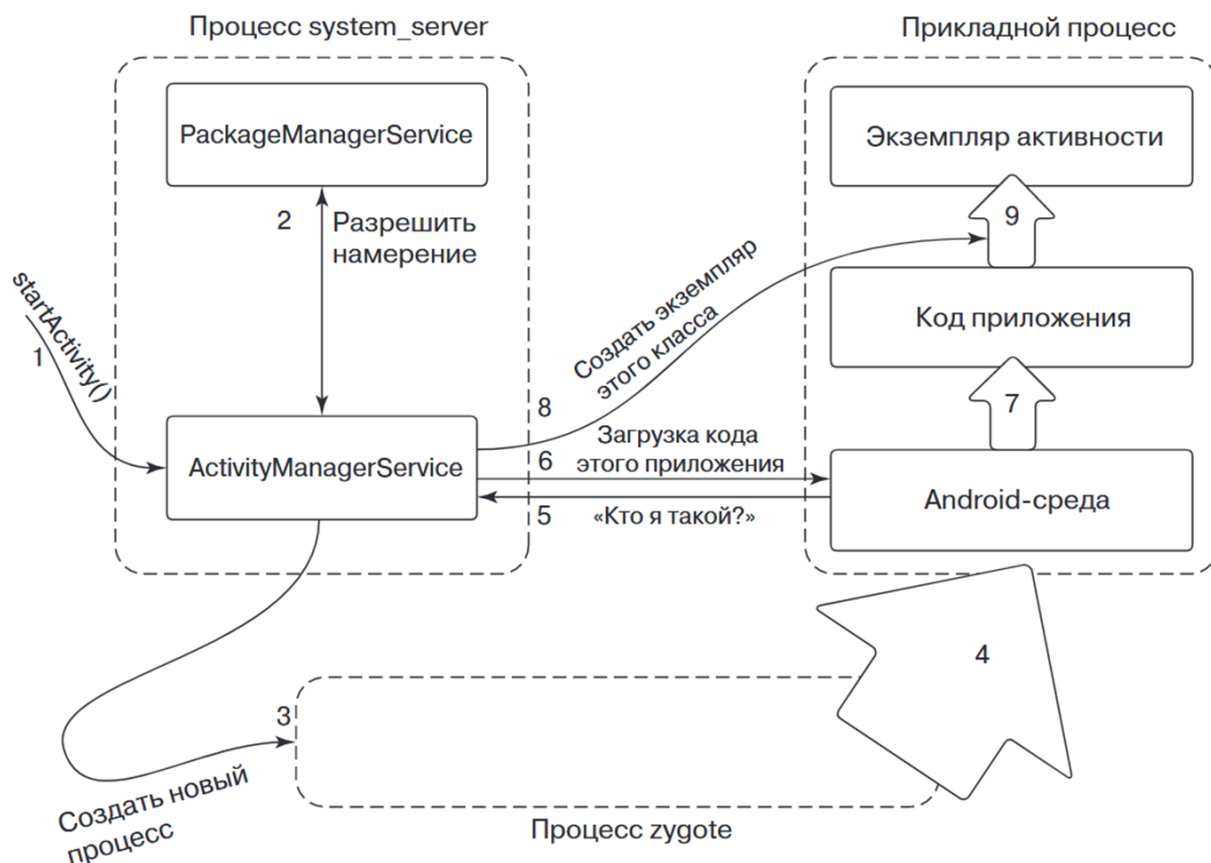


Рисунок 1.3 – Создание нового процесса

По умолчанию все компоненты одного приложения работают в одном процессе, и большинство приложений не должно менять это поведение. Однако, если необходимо контролировать, к какому процессу принадлежат определенный

компонент, возможно сделать это в файле манифеста.

Запись манифеста для каждого типа элементов компонента — `<activity>`, `<service>`, `<receiver>` и `<provider>` — поддерживает атрибут `android:process`, позволяющий задавать процесс, в котором следует выполнять этот компонент. Возможно установить данный атрибут так, чтобы каждый компонент выполнялся в собственном процессе, или так, чтобы только некоторые компоненты совместно использовали один процесс. Имеется возможность также настроить процесс `android:process` так, чтобы компоненты разных приложений выполнялись в одном процессе, при условии что приложения совместно используют один идентификатор пользователя Linux и выполняют вход с одним сертификатом.

1.3 Жизненный цикл процессов

Система Android сохраняет процесс приложения как можно дольше, но в конечном счете вынуждена удалять старые процессы, чтобы восстановить память для новых или более важных процессов. Чтобы определить, какие процессы сохранить, а какие удалить, система помещает каждый процесс в «иерархию важности» на основе компонентов, выполняющихся в процессе, и состояния этих компонентов. Процессы с самым низким уровнем важности исключаются в первую очередь, затем исключаются процессы следующего уровня важности и т.д., насколько это необходимо для восстановления ресурсов системы. Диспетчер активностей отвечает за определение того момента, когда процессы больше не нужны. Диспетчер отслеживает все активности, получатели, службы и поставщики контента, запущенные в процессе, в результате чего возможно определить, насколько важен (или неважен) тот или иной процесс.

Имеющийся в Android механизм устранения дефицита памяти, находящийся в ядре, использует показатель `oom_adj` (уровень важности) процесса для выстраивания четкого порядка, позволяющего определить, какие процессы он должен уничтожить. Диспетчер активностей отвечает за настройку показателей `oom_adj` всех процессов, которые соответствующим образом основаны на состоянии этих процессов, классифицируя их в основных категориях использования. Эти основные категории показаны в таблице 1.2, где сначала идут наиболее важные.

В последнем столбце показано типовое значение `oom_adj`, которое назначается процессу данного типа.

Таблица 1.2– Основные категории процессов и их значимость

Категория	Описание	<code>oom_adj</code>
SYSTEM	Системные процессы и демоны	-16
PERSISTENT	Постоянно работающие прикладные процессы	-12
FOREGROUND	Процессы, взаимодействующие в данный момент с пользователем	0
VISIBLE	Процессы, видимые ользователю	1
PERCEPTIBLE	Что-то, о чем знает пользователь	2
SERVICE	Запущенные фоновые службы	3
HOME	Главный (запускающий) процесс	4
CACHED	Неиспользуемый процесс	5

После того, когда уровень свободной оперативной памяти снизится, система настроит процессы таким образом, чтобы механизм устранения дефицита памяти сначала уничтожил кэшированные процессы, стараясь восстановить достаточный объем нужной оперативной памяти, затем главный процесс, процессы служб и далее вверх по списку.

Внутри конкретного уровня `oom_adj` механизм сначала уничтожит процессы, которые используют больше оперативной памяти, а затем перейдет к уничтожению тех, которые используют меньше памяти.

2 ПОТОКИ

Запущенное в Android приложение имеет собственный процесс и как минимум один поток. По умолчанию все компоненты одного приложения работают в одном процессе и потоке (называется «главным потоком»). Данный поток очень важен, так как он отвечает за диспетчеризацию событий на виджеты соответствующего интерфейса пользователя, включая события графического представления. Он также является потоком, в котором приложение взаимодействует с компонентами из набора инструментов пользовательского интерфейса Android (компонентами из пакетов `android.widget` и `android.view`). По существу, главный поток — это то, что иногда называют потоком пользовательского интерфейса.

Если в приложении есть какие-либо визуальные элементы, то в этом потоке запускается объект класса `Activity`, отвечающий за отображение на дисплее интерфейса (user interface, UI).

В главном `Activity` должно быть, как можно меньше вычислений, единственная его задача — отображать UI. Если главный поток будет занят вычислением сложных операций, то он потеряет связь с пользователем, `Activity` не сможет обрабатывать запросы пользователя и со стороны будет казаться, что приложение зависло. Если ожидание продолжается больше нескольких секунд,

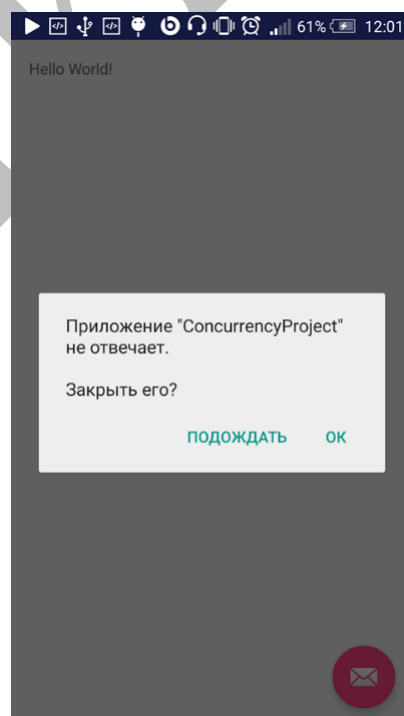


Рисунок 2.1 – Сообщение ANR

ОС Android отобразит сообщение ANR (application not responding — «приложение не отвечает») с предложением принудительно завершить приложение (рисунок.3.1). Для того, чтобы получить данное сообщение достаточно в главном потоке начать работу с файловой системой, сетью, криптографией и так далее.

2.1 Задание. Thread и Runnable

Создайте/переименуйте модуль thread. На экране требуется разместить элементы Button и TextView.

Посчитать в фоновом потоке среднее количество пар в день за период одного месяца. Общее количество пар и учебных дней вводятся в главном экране. Отобразить результат в TextView.

Главный поток создаётся автоматически, им можно управлять через объект класса Thread. Для этого требуется вызвать метод `currentThread()`, после чего возможно управлять потоком.

Класс Thread содержит несколько методов для управления потоками.

- `getName()` - получить имя потока
- `getPriority()` - получить приоритет потока
- `isAlive()` - определить, выполняется ли поток
- `join()` - ожидать завершения потока
- `run()` - запуск потока. В нём пишете свой код
- `sleep()` - приостановить поток на заданное время
- `start()` - запустить поток

Для получения информации о главном потоке и изменении его имени требуется добавить следующий код в MainActivity:

```
TextView infoTextView = findViewById(R.id.textView);
Thread mainThread = Thread.currentThread();
infoTextView.setText("Текущий поток: " + mainThread.getName());
// Меняем имя и выводим в текстовом поле
mainThread.setName("MireaThread");
infoTextView.append("\n Новое имя потока: " + mainThread.getName());
```

После запуска приложения на экране должно отобразиться имя потока до и после его изменения. Имя главного потока по умолчанию main, которое заменяется на MireaThread.

Для имитации вычислений в главном потоке требуется добавить следующий код в метод onClick, реагирующий на нажатие кнопки:

```
public void onClick(View view) {
    long endTime = System.currentTimeMillis() + 20 * 1000;

    while (System.currentTimeMillis() < endTime) {
        synchronized (this) {
            try {
                wait(endTime - System.currentTimeMillis());
            } catch (Exception e) {
            }
        }
    }
}
```

Запустите приложение. Произведите несколько нажатий на кнопку. Оцените работоспособность приложения.

В Linux (в том числе и в Android) у каждого потока исполнения есть собственный приоритет, который варьируется в пределах от -20 до 19, где меньшее число означает более высокий приоритет (рисунок 2.2).

Ядро Linux объединяет потоки в так называемые контрольные группы (cgroups) в зависимости от разных условий, таких, например, как видимость приложения на экране. Размещение потоков того или иного приложения в определенной группе автоматически накладывает на них ограничения. Так, потоки, помещенные в группу Background (то есть относящиеся к приложениям в фоне), получают всего 5% от общего процессорного времени.

-Combined dalvik/vm/Thread.c and -frameworks/base/include/utils/threads.h			
Thread.priority	Java name	Android property name	Unix priority
1	MIN_PRIORITY	ANDROID_PRIORITY_LOWEST,	19
2		ANDROID_PRIORITY_BACKGROUND + 6	16
3		ANDROID_PRIORITY_BACKGROUND + 3	13
4		ANDROID_PRIORITY_BACKGROUND	10
5	NORM_PRIORITY	ANDROID_PRIORITY_NORMAL	0
6		ANDROID_PRIORITY_NORMAL - 2	-2
7		ANDROID_PRIORITY_NORMAL - 4	-4
8		ANDROID_PRIORITY_URGENT_DISPLAY + 3	-5
9	MAX_PRIORITY	ANDROID_PRIORITY_URGENT_DISPLAY + 2	-6
10		ANDROID_PRIORITY_URGENT_DISPLAY	-8

Рисунок 2.2 – Шкала приоритета потоков

По умолчанию любые потоки одного приложения получают равный приоритет с основным потоком, а значит, могут влиять на его исполнение. Чтобы этого избежать, следует снизить приоритет фоновых потоков с помощью одного из двух методов:

Базовым классом для потоков в Android является класс `Thread`, который содержит необходимый функционал для создания потока. Но для того, чтобы выполнить внутри нового потока задачу, требуется использовать объект класса `Runnable`. `Thread`, получив объект этого класса, сразу же выполнит метод `run`.

```
new Thread(new Runnable() {  
    public void run() {  
        //do time consuming operations  
    }  
}).start();
```

`Thread.setPriority()` — принимает значения от 1 до 10, где 10 — самый высокий приоритет;

`Process.setThreadPriority()` — принимает стандартные для Android/Linux значения от -20 до 19.

Требуется перенести функционал, замедляющий работу приложения, в отдельный поток. Для этого создаётся экземпляр класса `Runnable`, у которого есть метод `run()`. Далее создаётся объект `Thread`, в конструкторе у которого указывается созданный `Runnable`. После этого возможно запускать новый поток с помощью метода `start()`. Перепишем пример.

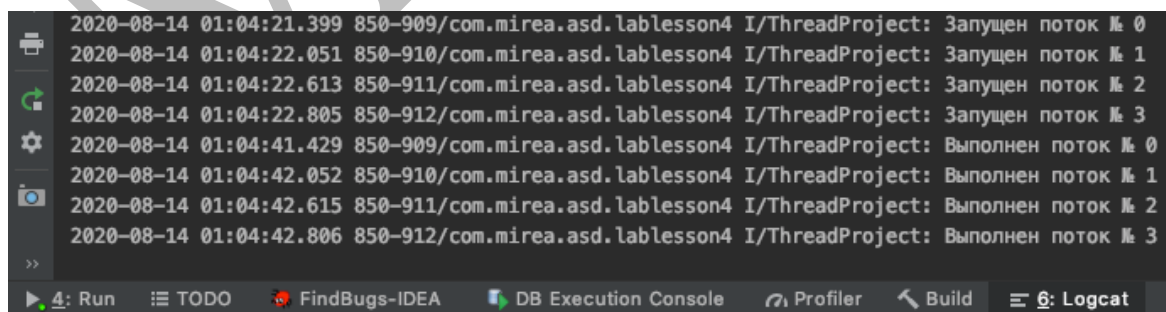
```

public class MainActivity extends AppCompatActivity {
    int counter = 0;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        TextView infoTextView = findViewById(R.id.textview);
        Thread mainThread = Thread.currentThread();
        infoTextView.setText("Текущий поток: " + mainThread.getName());
        // Меняем имя и выводим в текстовом поле
        mainThread.setName("MireaThread");
        infoTextView.append("\n Новое имя потока: " + mainThread.getName());
    }
    public void onClick(View view) {
        Runnable runnable = new Runnable() {
            public void run() {
                int numberThread = counter++;
                Log.i("ThreadProject", "Запущен поток № " + numberThread);
                long endTime = System.currentTimeMillis()
                    + 20 * 1000;

                while (System.currentTimeMillis() < endTime) {
                    synchronized (this) {
                        try {
                            wait(endTime -
                                System.currentTimeMillis());
                        } catch (Exception e) {
                        }
                    }
                }
                Log.i("ThreadProject", "Выполнен поток № " + numberThread);
            }
        };
        Thread thread = new Thread(runnable);
        thread.start();
    }
}

```

Замедляющий функционал перенесён в метод run() объекта Runnable. Теперь имеется возможность производить несколько нажатий по кнопке. Для демонстрации в код добавлено протоколирование логов Log.i(). При каждом нажатии создаётся новый поток, в котором выполняется код.



The screenshot shows the Logcat window with the following log entries:

Time	Process	Package	Class	Message
2020-08-14 01:04:21.399	850-909	com.mirea.asd.lablesson4	I/ThreadProject	Запущен поток № 0
2020-08-14 01:04:22.051	850-910	com.mirea.asd.lablesson4	I/ThreadProject	Запущен поток № 1
2020-08-14 01:04:22.613	850-911	com.mirea.asd.lablesson4	I/ThreadProject	Запущен поток № 2
2020-08-14 01:04:22.805	850-912	com.mirea.asd.lablesson4	I/ThreadProject	Запущен поток № 3
2020-08-14 01:04:41.429	850-909	com.mirea.asd.lablesson4	I/ThreadProject	Выполнен поток № 0
2020-08-14 01:04:42.052	850-910	com.mirea.asd.lablesson4	I/ThreadProject	Выполнен поток № 1
2020-08-14 01:04:42.615	850-911	com.mirea.asd.lablesson4	I/ThreadProject	Выполнен поток № 2
2020-08-14 01:04:42.806	850-912	com.mirea.asd.lablesson4	I/ThreadProject	Выполнен поток № 3

При такой организации сложно использовать полноценные возможности дополнительных потоков — нельзя ни поменять задачу, ни посмотреть результат вычислений. Хотя все это происходит в едином адресном пространстве, у Java-

разработчиков нет механизмов организации обмена ресурсами между соседними потоками.

Данный способ подходит только для операций, связанных с временем. Нет возможности: отслеживать жизненный цикл приложения, что приводит к утечкам памяти; обновлять графический интерфейс программы.

ИМПРЕА

3 ПЕРЕДАЧА ДАННЫХ МЕЖДУ ПОТОКАМИ

3.1 Задание. Thread в UI

Создать новый модуль. Название `data_thread`.

Основной поток также называют UI-поток. Именно в главном потоке возможно обновлять данные, отображаемые для пользователя. В создаваемых ранее потоках этого сделать нельзя. Существует несколько способов выполнять Runnable в UI-потоке. Это методы:

- `Activity.runOnUiThread(Runnable)`
- `View.post(Runnable)`
- `View.postDelayed(Runnable, long)`

В примере, приведённом ниже, создаётся 3 Runnable в методе `onCreate` и изменение экрана выполняется внутри метода `run`. Первые два метода похожи и отправляют Runnable на немедленную обработку. Третий метод позволяет указать задержку выполнения Runnable.


```

public class MainActivity extends AppCompatActivity {
    TextView tvInfo;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        tvInfo = findViewById(R.id.textView);
        final Runnable runn1 = new Runnable() {
            public void run() {
                tvInfo.setText("runn1");
            }
        };

        final Runnable runn2 = new Runnable() {
            public void run() {
                tvInfo.setText("runn2");
            }
        };

        final Runnable runn3 = new Runnable() {
            public void run() {
                tvInfo.setText("runn3");
            }
        };

        Thread t = new Thread(new Runnable() {
            public void run() {
                try {
                    TimeUnit.SECONDS.sleep(2);
                    runOnUiThread(runn1);
                    TimeUnit.SECONDS.sleep(1);
                    tvInfo.postDelayed(runn3, 2000);
                    tvInfo.post(runn2);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        t.start();
    }
}

```

Рассмотрите в какой последовательности происходит запуск процессов. Изучите методы `runOnUiThread`, `postDelayed`, `post`. В чём различия?

3.2 Задание. Looper

Создайте новый модуль. Название `looper`. В `main_activity.xml` требуется добавить `button` с методом `onClick`.

Одним из способов передачи данных из одного потока в другой — это создание очереди сообщений (`MessageQueue`) внутри потока (рисунок 3.1). В такую очередь возможно добавлять задания из других потоков, заданиями могут быть переменные, объекты или участок кода для исполнения (`Runnable`).

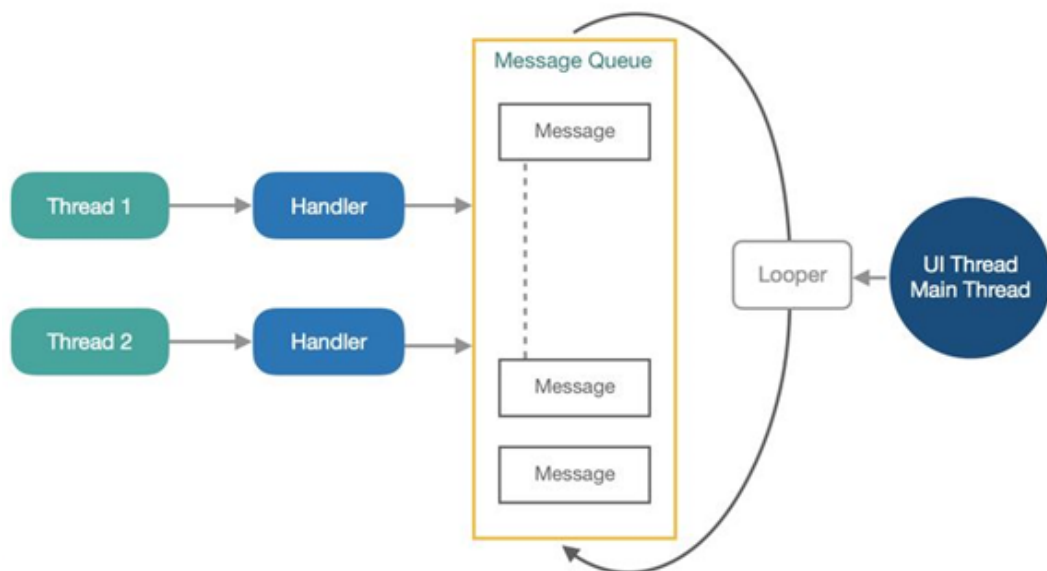


Рисунок 3.1 – Механизм функционирования Looper

Для организации очереди, требуется воспользоваться классами Handler и Looper: первый отвечает за организацию очереди, а второй в бесконечном цикле проверяет, нет ли в ней новых задач для потока. Поток работает, пока связанный с ним Looper не будет остановлен. Для создания класса требуется вызвать контекстное меню, нажав на директорию с кодом в папке java : New->File->Выбрать папку ...src/main/java->Ввести имя файла->Установите родительским классом Thread

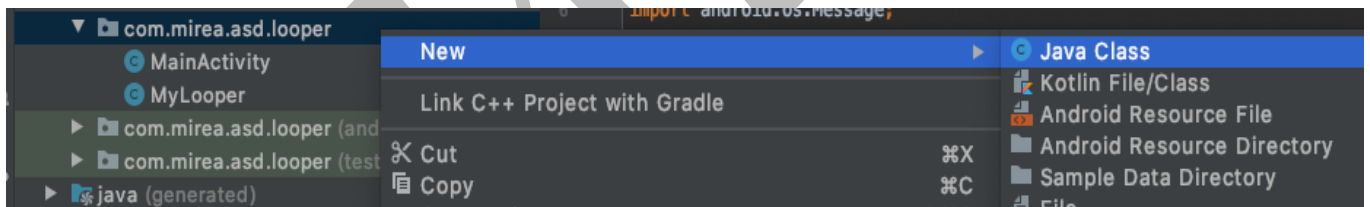


Рисунок 3.2 - Создание нового класса

```

public class MyLooper extends Thread {
    private int number = 0;
    Handler handler;

    @SuppressWarnings("HandlerLeak")
    @Override
    public void run(){
        Log.d("MyLooper", "run");
        Looper.prepare();
        handler = new Handler(){
            @Override
            public void handleMessage(Message msg){
                Log.d("MyLooper", number + ":" + msg.getData().getString("KEY"));
                number++;
            }
        };
        Looper.loop();
    }
}

```

Для создания *Looper*, вызывается статический метод *Looper.prepare()*. Созданный *Looper* будет связан с потоком, в котором вызван этот метод. Для старта *Looper* используется статический метод *Looper.loop()*. Между вызовами методов *prepare()* и *loop()* как правило создается *Handler*, который будет обрабатывать сообщения, приходящие в *MessageQueue*.

Запуск потока осуществляется по следующей схеме — создание нового объекта и вызов метода *start* в методе *onCreate*:

```
myLooper = new MyLooper();  
myLooper.start();
```

После выполнения данного метода создаётся новый поток. Это означает, что инициализация переменных и создание объектов будут уже идти параллельно с теми вызовами, которые установлены в следующих строчках после команды *myLooper.start()*. Поэтому перед обращением к очереди в новом потоке требуется проверять объект *handler* на существование. Следующий код отправляет слово *mirea* в другой поток по нажатию на кнопку:

```
public void onClick(View view) {  
    Message msg = new Message();  
    Bundle bundle = new Bundle();  
    bundle.putString("KEY", "mirea");  
    msg.setData(bundle);  
    if (myLooper != null) {  
        myLooper.handler.sendMessage(msg);  
    }  
}
```

В данном примере создается объект класса *Message* в который передаются объект *Bundle* с передаваемыми значениями. Данное сообщение передаётся в *MessageQueue* через метод *sendMessage*.

Реализуйте пример, в котором отображается Ваш возраст и кем Вы работаете. Количество лет соответствует времени задержки. Результат вычисления осуществлять через *Log.d*.

3.3 Задание. Loaders

Создайте новый модуль. Название *LoaderManger*.

Основная задача 90% всех мобильных приложений — это быстро и незаметно для пользователя загрузить данные из сети или файловой системы, а затем

отобразить их на дисплее.

Загрузчики появились в Android 3.0, упрощают асинхронную загрузку данных в операцию или фрагмент. Загрузчики имеют следующие свойства:

- применяются для любых операций [Activity](#) и [Fragment](#);
- обеспечивают асинхронную загрузку данных;
- отслеживают источник своих данных и выдают новые результаты при изменении контента;
- автоматически переподключаются к последнему курсору загрузчика при воссоздании после изменения конфигурации. Таким образом, им не требуется повторно запрашивать свои данные.

Лoader – это компонент Android, который через класс LoaderManager связан с жизненным циклом Activity и Fragment. Это позволяет использовать их без опасения, что данные будут утрачены при закрытии приложения или результат вернется не в тот коллбэк.

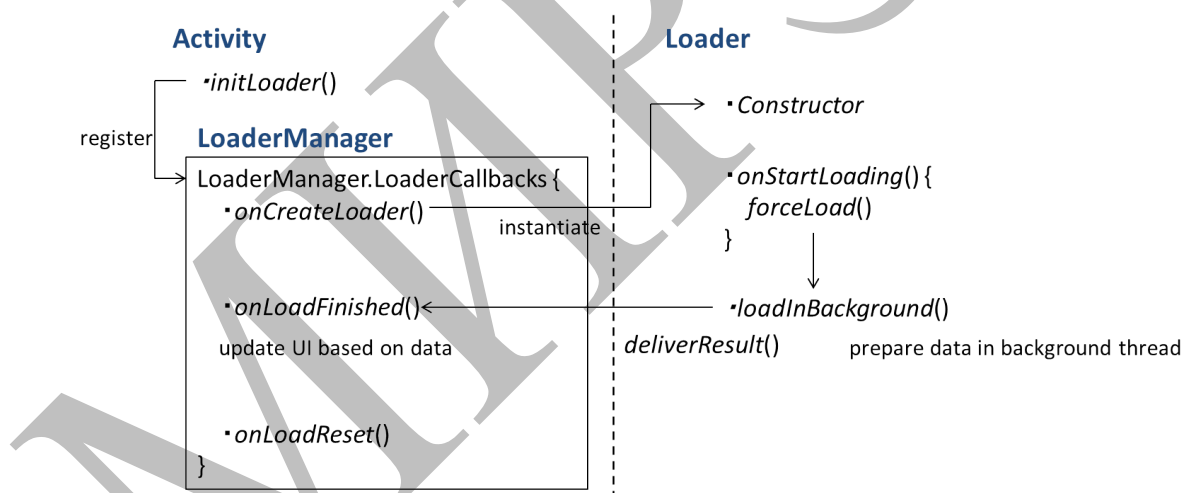


Рисунок 3.3 – Отношение между Activity и LoaderManager

Каждая активность и каждый фрагмент имеет один экземпляр менеджера LoaderManager, который работает с загрузчиками через методы `initLoader()`, `restartLoader()`, `destroyLoader()`. Активность через данного менеджера может предупредить о своём уничтожении, чтобы LoaderManager в свою очередь закрыл загрузчики для экономии ресурсов.

Сам LoaderManager не знает, как данные загружаются в приложение. Он выдаёт указания загрузчику начать, остановить, обновить загрузку данных и другие

команды.

Таблица 3.1 - Информация об API-интерфейсе загрузчика

Класс/интерфейс	Описание
LoaderManager	Абстрактный класс, связываемый с Activity или Fragment для управления одним или несколькими интерфейсами Loader. Это позволяет приложению управлять длительно выполняющимися операциями вместе с жизненным циклом Activity или Fragment; чаще всего этот класс используется с CursorLoader, однако приложения могут писать свои собственные загрузчики для работы с другими типами данных. Имеется только один класс LoaderManager на операцию или фрагмент. Однако у класса LoaderManager может быть несколько загрузчиков.
LoaderManager.LoaderCallbacks	Интерфейс обратного вызова, обеспечивающий взаимодействие клиента с LoaderManager. Например, с помощью метода обратного вызова onCreateLoader() создается новый загрузчик.
Loader	Абстрактный класс, который выполняет асинхронную загрузку данных. Это базовый класс для загрузчика. Обычно используется CursorLoader, но можно реализовать и собственный подкласс. Когда загрузчики активны, они должны отслеживать источник своих данных и выдавать новые результаты при изменении контента.
AsyncTaskLoader	Абстрактный загрузчик, который предоставляет AsyncTask для выполнения работы.
CursorLoader	Подкласс класса AsyncTaskLoader , который запрашивает ContentResolver и возвращает Cursor . Этот класс реализует протокол Loader стандартным способом для выполнения запросов к курсорам. Он строится на AsyncTaskLoader для выполнения запроса к курсору в фоновом потоке, чтобы не блокировать пользовательский интерфейс приложения. Использование этого загрузчика — это лучший способ асинхронной загрузки данных из ContentProvider вместо выполнения управляемого запроса через платформу или API-интерфейсы операции.

Приведенные в таблице 3 классы и интерфейсы являются наиболее важными компонентами, с помощью которых в приложении реализуется загрузчик. При создании каждого загрузчика не нужно использовать все эти компоненты, однако всегда следует указывать ссылку на LoaderManager для инициализации загрузчика и использовать реализацию класса Loader, например CursorLoader.

LoaderManager работает с объектами Loader<D>, где D является контейнером для загружаемых данных. При этом данные не обязательно должны быть курсором. Это могут быть и List, JSONArray и т.д. В одной активности может быть несколько загрузчиков, которые являются объектами.

Класс Loader является общим. Также доступны специализированные загрузчики AsyncTaskLoader и CursorLoader.

Работа с LoaderManager происходит через три метода обратного вызова интерфейса LoaderManager.LoaderCallbacks<D>.

```
public class MainActivity extends AppCompatActivity implements
LoaderManager.LoaderCallbacks<D>{

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

    @NonNull
    @Override
    public Loader<D> onCreateLoader(int id, @Nullable Bundle args) {
        return null;
    }

    @Override
    public void onLoadFinished(@NonNull Loader<D> loader, D data) {

    }

    @Override
    public void onLoaderReset(@NonNull Loader<D> loader) {

    }
}
```

- Метод onCreateLoader() возвращает новый загрузчик. LoaderManager вызывает метод при создании Loader. При попытке доступа к загрузчику (например, посредством метода initLoader()), выполняется проверка, существует ли загрузчик, указанный с помощью идентификатора. Если он не существует, он вызывает метод onCreateLoader(). Именно здесь и создаётся новый загрузчик.

- Метод onLoadFinished вызывается автоматически, когда Loader завершает загрузку данных. Загрузчик следит за поступающими данными, а менеджер получает уведомление о завершении загрузки и передаёт результат данному методу. Этот метод гарантировано вызывается до высвобождения последних данных, которые были предоставлены этому загрузчику. К этому моменту необходимо полностью перестать использовать старые данные (поскольку они скоро будут заменены). Однако этого не следует делать самостоятельно, поскольку данными владеет загрузчик и он позаботится об этом. Загрузчик высвободит данные, как только узнает, что приложение их больше не использует. Например, если данными

является курсор из `CursorLoader`, не следует вызывать `close()` самостоятельно. Если курсор размещается в `CursorAdapter`, следует использовать метод `swapCursor()` с тем, чтобы старый `Cursor` не закрылся.

– Метод `onLoadReset()` перезагружает данные в загрузчике. Этот метод вызывается, когда состояние созданного ранее загрузчика сбрасывается, в результате чего его данные теряются. Этот обратный вызов позволяет узнать, когда данные вот-вот будут высвобождены, с тем чтобы можно было удалить свою ссылку на них.

Для задач асинхронной загрузки данных в отдельном потоке используется класс, наследующий `AsyncTaskLoader<D>` вместо `Loader<D>`. Класс `AsyncTaskLoader<D>` является абстрактным и работает как `AsyncTask`. На основе этого класса возможно реализовать абстрактный метод `loadInBackground()`. Пример реализации представлен на рисунке 3.5.

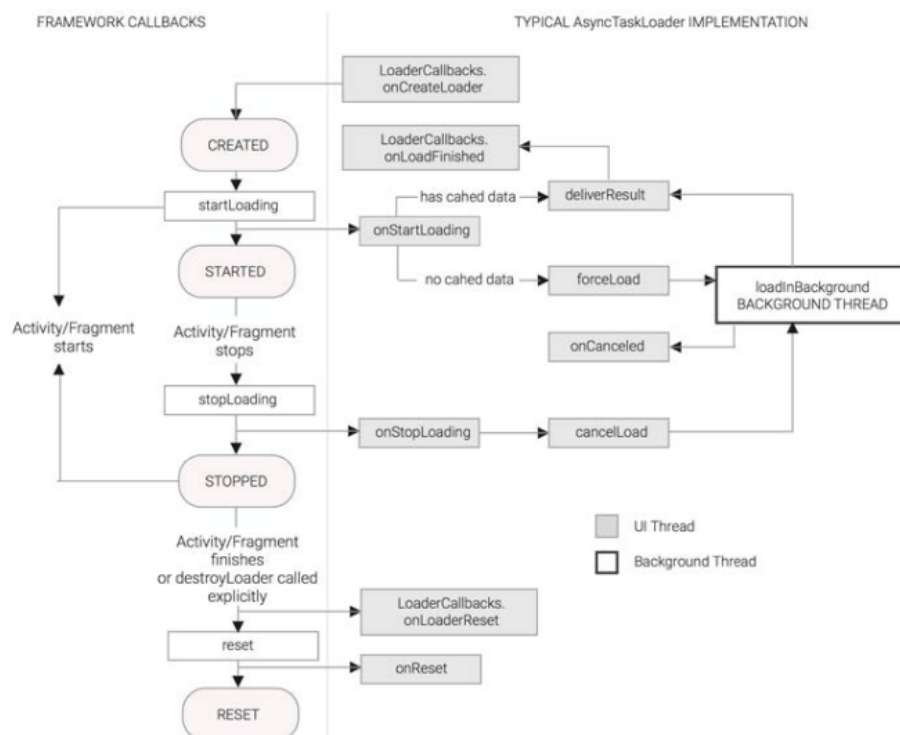


Рисунок 3.4 – Механизм `AsyncTaskLoader`

Слушатель получает информацию от загрузчика. Для этого менеджер регистрирует слушатель `OnLoadCompleteListener<D>`, который прослушивает события. Когда загрузка закончилась, то вызывается метод `onLoadFinished(Loader<D> loader, D result)`.

Чтобы загрузчик начал работать, его требуется запустить. Запущенный загрузчик следит за данными, пока его не перезапустят или остановят.

Остановленный загрузчик продолжает мониторить изменения в данных, но не сообщает о них. При необходимости требуется заново запустить или перезапустить остановленный загрузчик. При перезапуске загрузчик не должен запускать новую загрузку данных и отслеживать изменения. Его задача - освободить лишние данные. Это состояние редко используется, но в некоторых случаях оно нужно.

Основная задача сводится к созданию собственного загрузчика, реализации метода `loadInBackground()` и переопределении методов `onStartLoading()`, `onStopLoading()`, `onReset()`, `onCanceled()`, `deliverResult(D results)`.

Требуется создать класс `Loader` (New->Java Class->...):

```
public class MyLoader extends AsyncTaskLoader<String> {
    private String firstName;
    public static final String ARG_WORD = "word";

    public MyLoader(@NonNull Context context, Bundle args) {
        super(context);
        if (args != null)
            firstName = args.getString(ARG_WORD);
    }

    @Override
    protected void onStartLoading() {
        super.onStartLoading();
        forceLoad();
    }

    @Override
    public String loadInBackground() {
        // emulate long-running operation
        SystemClock.sleep(5000);
        return firstName;
    }
}
```

В методе `loadInBackground()` – требуется загрузить данные `String` – в данном примере это контейнер для данных (`String`), которые загружает `Loader`. Тут может размещаться запрос к базе данных, сетевой части и т.д. В данном случае возвращаются данные, которые указаны при создании объекта.

У `Loader` имеется два метода с одинаковой сигнатурой:

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

где `int id` - идентификатор `Loader` для различения их между собой;

`Bundle args` - класс `Bundle`, с помощью которого передаются аргументы для создания `Loader`;

LoaderManager.LoaderCallbacks – предназначен для создания Loader и получения от него результата работы:

```
public class MainActivity extends AppCompatActivity implements
    LoaderManager.LoaderCallbacks<String> {
    public final String TAG = this.getClass().getSimpleName();
    private int LoaderID = 1234;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Bundle bundle = new Bundle();
        bundle.putString(MyLoader.ARG_WORD, "mirea");
        getSupportLoaderManager().initLoader(LoaderID, bundle, this);
    }

    @Override
    public void onLoaderReset(@NonNull Loader<String> loader) {
        Log.d(TAG, "onLoaderReset");
    }

    @NonNull
    @Override
    public Loader<String> onCreateLoader(int i, @Nullable Bundle bundle) {
        if (i == LoaderID) {
            Toast.makeText(this, "onCreateLoader:" + i, Toast.LENGTH_SHORT).show();
            return new MyLoader(this, bundle);
        }
        return null;
    }

    @Override
    public void onLoadFinished(@NonNull Loader<String> loader, String s) {
        if (loader.getId() == LoaderID) {
            Log.d(TAG, "onLoadFinished" + s);
            Toast.makeText(this, "onLoadFinished:" + s, Toast.LENGTH_SHORT).show();
        }
    }
}
```

После запуска приложения отобразится всплывающее сообщение о начале работы нового потока с указанием идентификатора и через 5 секунд окончание работы.

Добавьте на экран EditText и Button. На основе данного примера создайте приложение, перемешивающее символы из поля EditText, введенные пользователем. Данные должны обрабатываться в Loader, возвращать новую строку и устанавливать новое значение в TextView.

Внимание!

На смену данному компоненту пришли архитектурные компоненты ViewModel и LiveData, которые будут рассмотрены на другом практическом занятии.

4 SERVICE

Service является компонентом приложения, который может выполнять длительные операции в фоновом режиме и не содержит пользовательского интерфейса. Другой компонент приложения может запустить службу, которая продолжит работу в фоновом режиме даже в том случае, когда пользователь перейдет в другое приложение. Кроме того, компонент может привязаться к службе для взаимодействия с ней и даже выполнять межпроцессное взаимодействие (IPC). Например, служба может обрабатывать сетевые транзакции, воспроизводить музыку, выполнять ввод-вывод файла или взаимодействовать с поставщиком контента, и все это в фоновом режиме.

Фактически служба может принимать две формы:

- запущенная, когда компонент приложения (например, операция) запускает ее вызовом [startService\(\)](#). После запуска служба может работать в фоновом режиме в течение неограниченного времени, даже если уничтожен компонент, который ее запустил. Обычно запущенная служба выполняет одну операцию и не возвращает результатов вызывающему компоненту. Например, она может загружать или выгружать файл по сети. Когда операция выполнена, служба должна остановиться самостоятельно.
- Привязанная, когда компонент приложения привязывается к ней вызовом [bindService\(\)](#). Привязанная служба предлагает интерфейс клиент-сервер, который позволяет компонентам взаимодействовать со службой, отправлять запросы, получать результаты и даже делать это между разными процессами посредством межпроцессного взаимодействия (IPC). Привязанная служба работает только пока к ней привязан другой компонент приложения. К службе могут быть привязаны несколько компонентов одновременно, но когда все они отменяют привязку, служба уничтожается.

Что лучше использовать — службу или поток?

Служба — это компонент, который может выполняться в фоновом режиме, даже когда пользователь не взаимодействует с приложением. Следовательно, вы

должны создавать службу только в том случае, если вам нужно именно это.

Если вам требуется выполнить работу за пределами основного потока, но только в то время, когда пользователь взаимодействует с приложением, то вам, вероятно, следует создать новый поток, а не службу. Например, если вы хотите воспроизводить определенную музыку, но только во время работы операции, вы можете создать поток в `onCreate()`, запустить его выполнение в методе `onStart()`, а затем остановить его в методе `onStop()`.

Помните, что если Вы действительно используете службу, она выполняется в основном потоке вашего приложения по умолчанию, поэтому вы должны создать новый поток в службе, если она выполняет интенсивные или блокирующие операции.

Все сервисы наследуются от класса `Service` и проходят следующие этапы жизненного цикла:

- метод `onCreate()` вызывается при создании сервиса;
- метод `onStartCommand()` вызывается при получении сервисом команды, отправленной с помощью метода `startService()`;
- метод `onBind()` вызывается при закреплении клиента за сервисом с помощью метода `bindService()`;
- метод `onDestroy()` вызывается при завершении работы сервиса

4.1 Создание Service

Чтобы определить службу, необходимо создать новый класс, расширяющий базовый класс `Service` возможно воспользоваться готовым мастером создания класса для сервиса в Android Studio. Требуется вызвать контекстное меню на папке `java` (или на имени пакета) и выбираем `New | Service | Service` (рисунок 4.1):

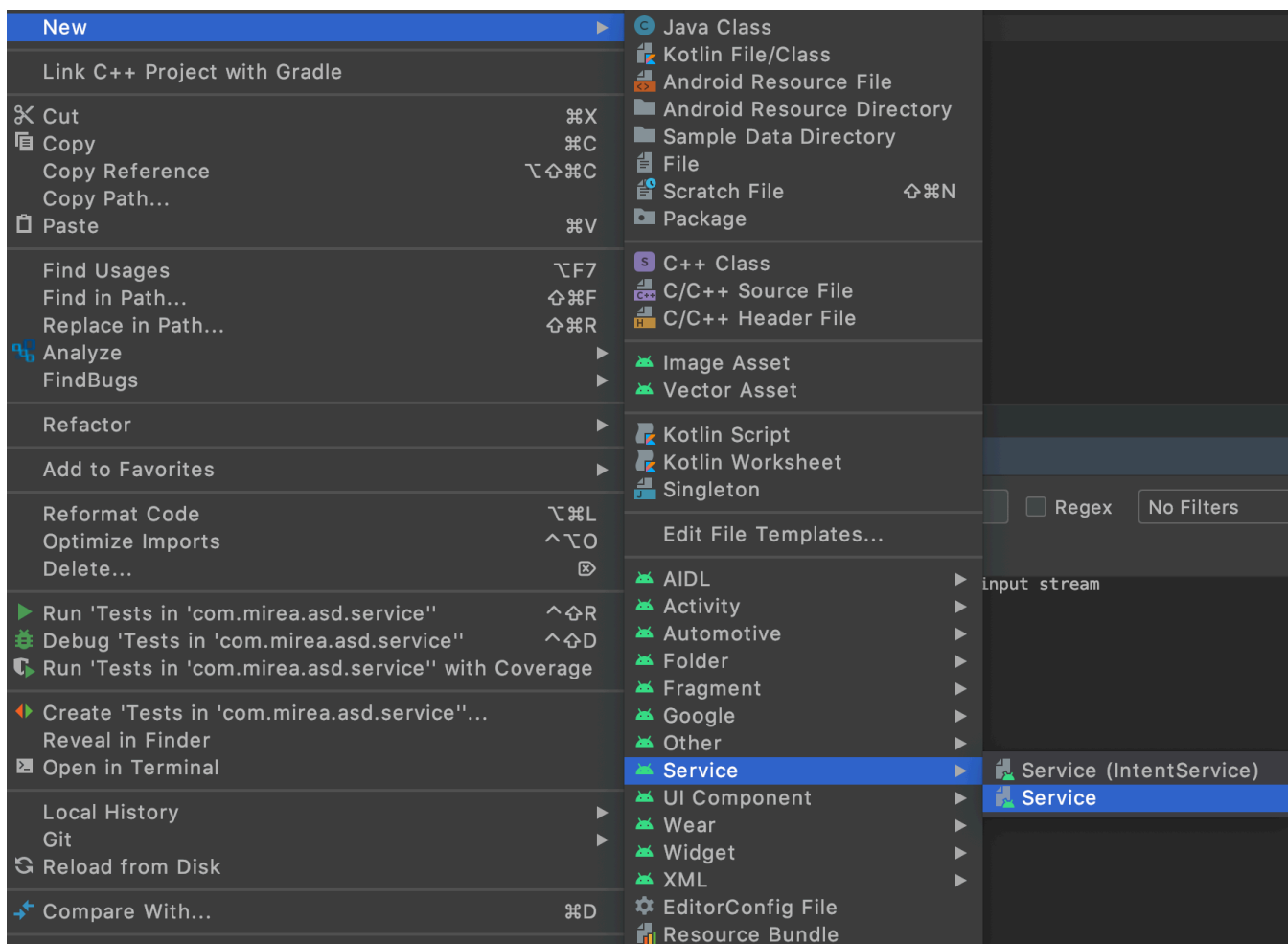


Рисунок 4.1 – Создание Service

В следующем окне производится выбор имени сервиса. Атрибут *exported* даёт возможность другим приложениям получить доступ к вашему сервису. Имеются и другие атрибуты, например, *permission*, чтобы сервис запускался только вашим приложением. Установите имя сервиса `PlayerService`.

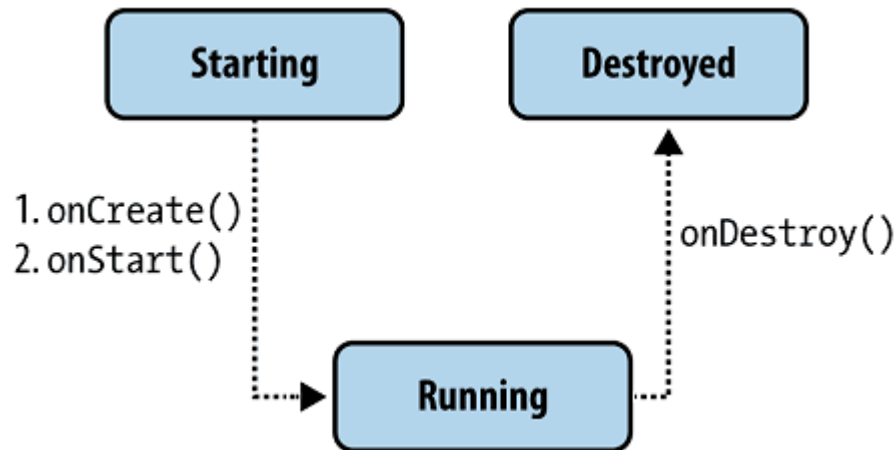
```
public class MyService extends Service {
    @Override
    public IBinder onBind(Intent intent) {
        // TODO: Return the communication channel to the service.
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

Требуется убедиться, что в манифест-файле добавлена запись о сервисе в секции `<application>`.

```
<service
    android:name=".MyService"
    android:enabled="true"
    android:exported="true" />
```

4.2 Жизненный цикл служб

Подобно активностям служба имеет свои методы жизненного цикла:



Для быстрого создания заготовок нужных методов используются команды меню Code | Override Methods или производится набор имени метода, используя автодополнение.

Реализуя данные методы обратного вызова в своей службе, имеется возможность контролировать жизненные циклы службы. В полном жизненном цикле службы существует два вложенных цикла:

- полная целая жизнь службы — промежуток между временем вызова метода `onCreate()` и временем возвращения `onDestroy()`. Подобно активности, для служб производят начальную инициализацию в `onCreate()` и освобождают все остающиеся ресурсы в `onDestroy()`
- активная целая жизнь службы — начинается с вызова метода `onStartCommand()`. Этому методу передается объект `Intent`, который передавался в метод `startService()`.

Из своего приложения службу возможно запустить вызовом метода `Context.startService()`, остановить через `Context.stopService()`. Служба может остановить сама себя, вызывая методы `Service.stopSelf()` или `Service.stopSelfResult()`.

Возможно установить подключение к работающей службе и использовать это подключение для взаимодействия со службой. Подключение устанавливают вызовом метода `Context.bindService()` и закрывают вызовом `Context.unbindService()`. Если служба уже была остановлена, вызов метода `bindService()` может её запустить.

Методы `onCreate()` и `onDestroy()` вызываются для всех служб независимо от того, запускаются ли они через `Context.startService()` или `Context.bindService()`.

4.3 Запуск сервиса и управление его перезагрузкой

В большинстве случаев также необходимо переопределить метод `onStartCommand()`. Он вызывается каждый раз, когда сервис стартует с помощью метода `startService()`, поэтому может быть выполнен несколько раз на протяжении работы.

Метод `onStartCommand()` позволяет указать системе, каким образом обрабатывать перезапуск, если сервис остановлен системой без явного вызова методов `stopService()` или `stopSelf()`.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId){
    return START_STICKY;
}
```

Службы запускаются в главном потоке приложения, что означает, что любые операции, выполняющиеся в обработчике `onStartCommand()`, будут работать в контексте главного потока GUI. На практике при реализации сервиса в методе `onStartCommand()` создают и запускают новый поток, чтобы выполнять операции в фоновом режиме и останавливать сервис, когда работа завершена.

Такой подход позволяет методу `onStartCommand()` быстро завершить работу и контролировать поведение сервиса при его повторном запуске, используя одну из констант.

- `START_STICKY` - описывает стандартное поведение. Если используется это значение, обработчик `onStartCommand()` будет вызываться при повторном запуске сервиса после преждевременного завершения работы. Стоит обратить внимание, что аргумент `Intent`, передаваемый в `onStartCommand()`, получит значение `null`. Данный режим обычно используется для служб, которые сами обрабатывают свои состояния, явно стартуя и завершая свою работу при необходимости (с помощью методов `startService()` и `stopService()`). Это относится к службам, которые проигрывают музыку или выполняют другие задачи в фоновом режиме

- `START_NOT_STICKY` - режим используется в сервисах, которые

запускаются для выполнения конкретных действий или команд. Как правило, такие службы используют `stopSelf()` для прекращения работы, как только команда выполнена. После преждевременного прекращения работы службы, работающие в данном режиме, повторно запускаются только в том случае, если получают вызовы. Если с момента завершения работы сервиса не был запущен метод `startService()`, он остановится без вызова обработчика `onStartCommand()`. Данный режим идеально подходит для сервисов, которые обрабатывают конкретные запросы, особенно это касается регулярного выполнения заданных действий (например, обновления или выполнения сетевых запросов). Вместо того, чтобы перезапускать сервис при нехватке ресурсов, часто более целесообразно позволить ему остановиться и повторить попытку запуска по прошествии запланированного интервала

- `START_REDELIVER_INTENT` - в некоторых случаях требуется убедиться, что команды, которые отправляются сервису, выполнены. Этот режим — комбинация предыдущих двух. Если система преждевременно завершила работу сервиса, он запустится повторно, но только когда будет сделан явный запрос на запуск или если процесс завершился до вызова метода `stopSelf()`. В последнем случае вызовется обработчик `onStartCommand()`, он получит первоначальное намерение, обработка которого не завершилась должным образом.

5 WORKMANAGER

На Google I/O 2018, Google анонсировали библиотеку WorkManager, являющуюся рекомендованным способом для управления задачами, которые должны выполняться не в UI – потоке, даже когда пользователь уже активно не взаимодействует с приложением.

Задачи, эффективно решаемые с помощью WorkManager

- Выполнение сетевых запросов с заданной периодичностью;
- Очищение кэша базы данных раз в сутки;
- Обновление информации для виджетов;
- Последовательное выполнение фоновых задач;
- Скачивание тяжёлых файлов.

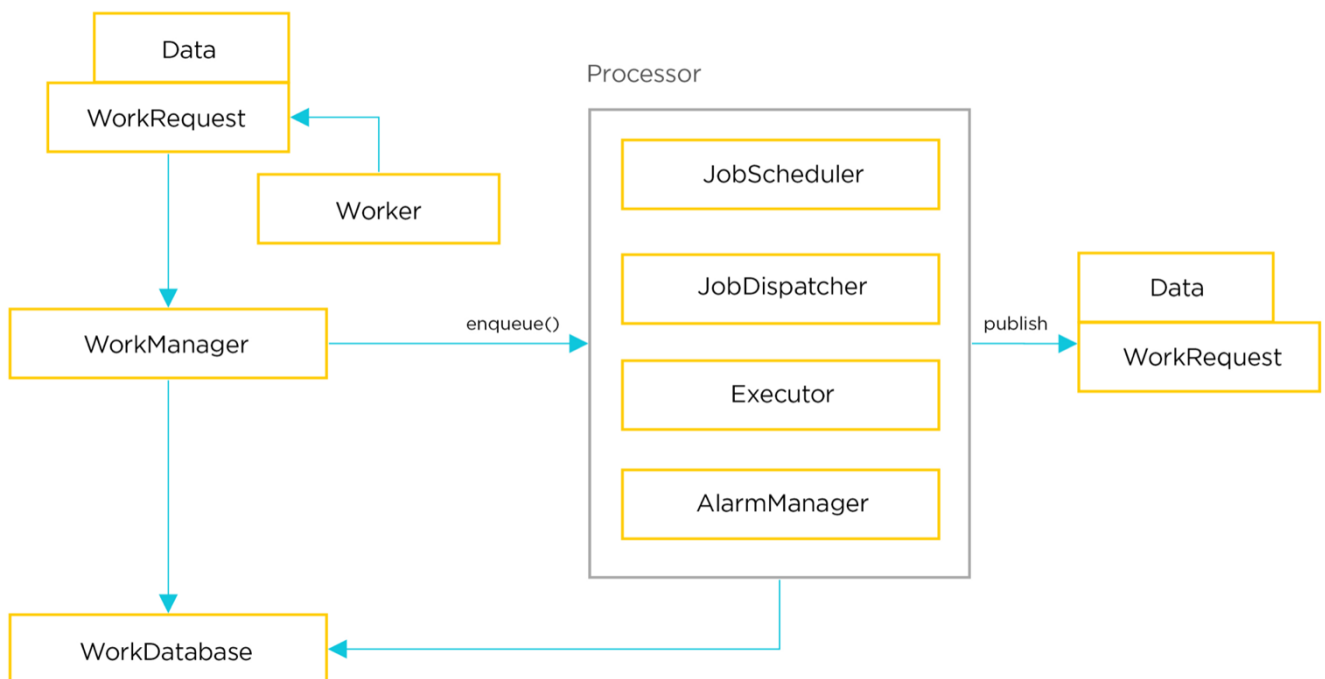
С каждым выходом новой версии Android, разработчики операционной системы всё больше и больше уделяли внимание оптимизации времени работы батареи телефона.

Начиная с Android 6.0 Marshmallow разработчики Google предложили так называемый Doze mode – то есть такой режим, в котором если устройство лежит без движения и без зарядки, спустя час оно переходит в специальный режим, когда почти все приложения перестают выполнять какую-то фоновую работу и потреблять батарею. Кроме того, начиная с Android 8.0 Oreo работа сервисов также была ограничена.

Таким образом, до 2018 года, для выполнения работы в фоне разработчику необходимо было разбираться во всех деталях работы фоновых задач для каждой версии операционной системы. WorkManager облегчает такие задачи и под капотом имеет реализацию, основанную на базе JobScheduler, Firebase JobDispatcher, Alarm Manager + Broadcast receivers. В зависимости от версии операционной системы, доступности Google Play сервисов на телефоне WorkManager выберет подходящую реализацию и выполнит задачу. Кроме того, что вам теперь не нужно продумывать все нюансы реализации работы фоновых задач, WorkManager имеет удобный интерфейс, скрывающий все сложности, позволяя разработчику сконцентрироваться на бизнес-логике задачи.

Основные классы

- **WorkManager** – главный класс, который будет передавать в работу логику на выполнение через **WorkRequest**
- **Worker** – класс, наследником которого должен быть собственный класс в котором нужно определить логику работы фоновой задачи (например сохранение данных в БД, запрос в сеть, загрузка данных и т.д).
- **WorkRequest** – класс, необходимы для описания критериев запуска задачи (например подключен ли Wi-Fi или достаточно ли заряда батареи). Кроме того, через класс **WorkRequest** нужно указать тип задачи – разовая (**OneTimeWorkRequest**) или повторяющаяся (**PeriodicWorkRequest**) – например делать запрос в сеть каждые 30 минут. Период для повторяющихся задач можно гибко настраивать – об этом мы поговорим чуть позже.
- **WorkStatus** – пригодится если нужно узнать статус задачи (running, enqueued, finished) конкретного **WorkRequest**



5.1 Задание

Создать модуль **WorkManager**. Добавить в пример критерии запуска: напр. Наличие интернета.

В файл *build.gradle(Module ...)* в раздел *dependencies* требуется добавить

библиотеку для работы с worker:

```
dependencies {  
    ...  
    implementation 'androidx.work:work-runtime:2.5.0'  
    ...  
}
```

В модуле создать класс с родительским классом Worker

```
public class UploadWorker extends Worker {  
    static final String TAG = "UploadWorker";  
  
    public UploadWorker(  
        @NonNull Context context,  
        @NonNull WorkerParameters params) {  
        super(context, params);  
    }  
  
    @Override  
    public Result doWork() {  
        Log.d(TAG, "doWork: start");  
        try {  
            TimeUnit.SECONDS.sleep(10);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
  
        Log.d(TAG, "doWork: end");  
        return Result.success();  
    }  
}
```

В MainActivity добавить вызов Worker:

```
public class MainActivity extends AppCompatActivity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        WorkRequest uploadWorkRequest =  
            new OneTimeWorkRequest.Builder(UploadWorker.class)  
                .build();  
        WorkManager  
            .getInstance(this)  
            .enqueue(uploadWorkRequest);  
    }  
}
```

6 КОНТРОЛЬНОЕ ЗАДАНИЕ

В созданном ранее проекте MireaProject требуется добавить функционал, основной задачей которого будет являться воспроизведение аудиофайла, добавленного ранее в ресурсы. В новом фрагменте разместить button для воспроизведения и остановки музыкальных композиций/композиции. Придумать собственный дизайн данного проигрывателя.

Далее приведён пример решения данной задачи на основе Service.

Аудиофайл должен находиться в `res/raw`, поэтому следует создать требуемую папку в проекте. Вызвать контекстное меню у папки `res|New|Android Resource Directory`(рисунок.5.1).

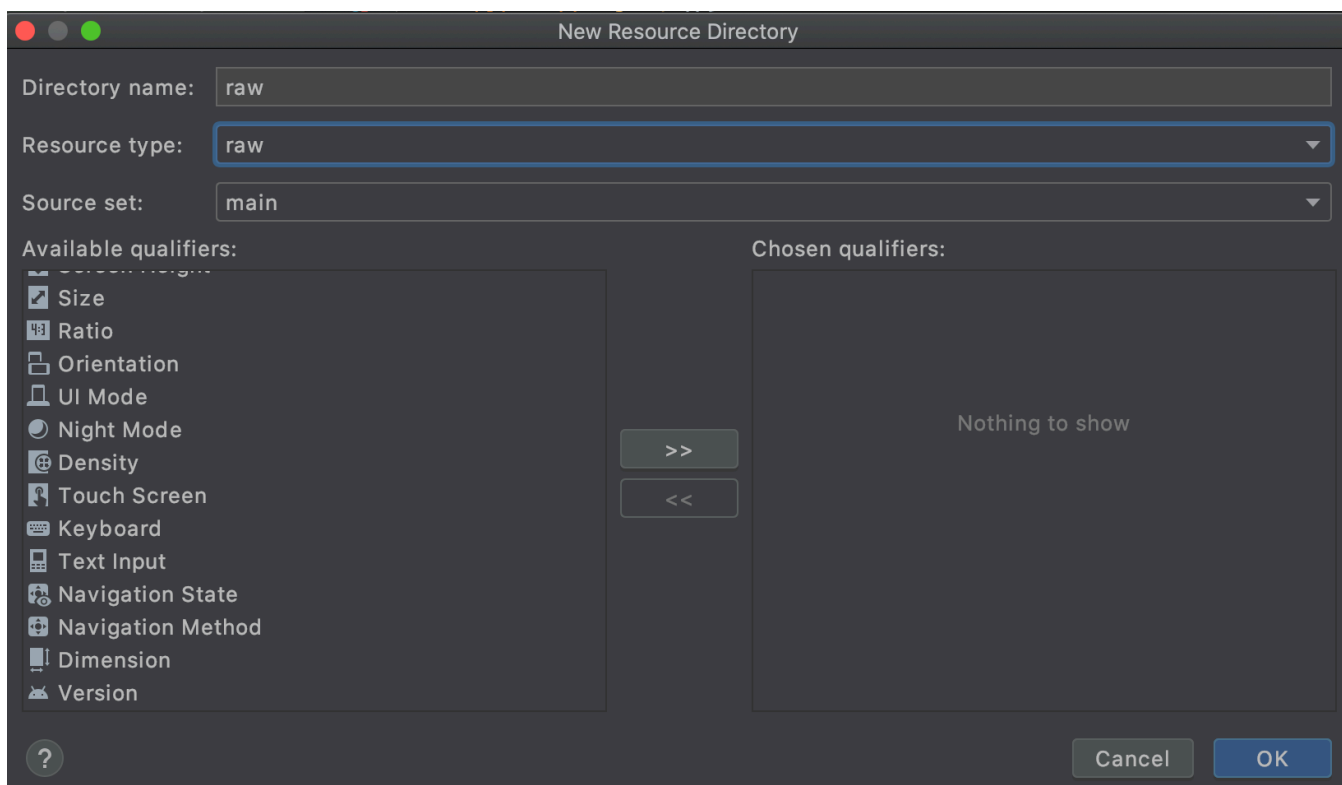


Рисунок 6.1 – Создание папки для хранения медиа-файлов

После этого требуется скачать любой файл `.mp3` и скопировать в данную папку:

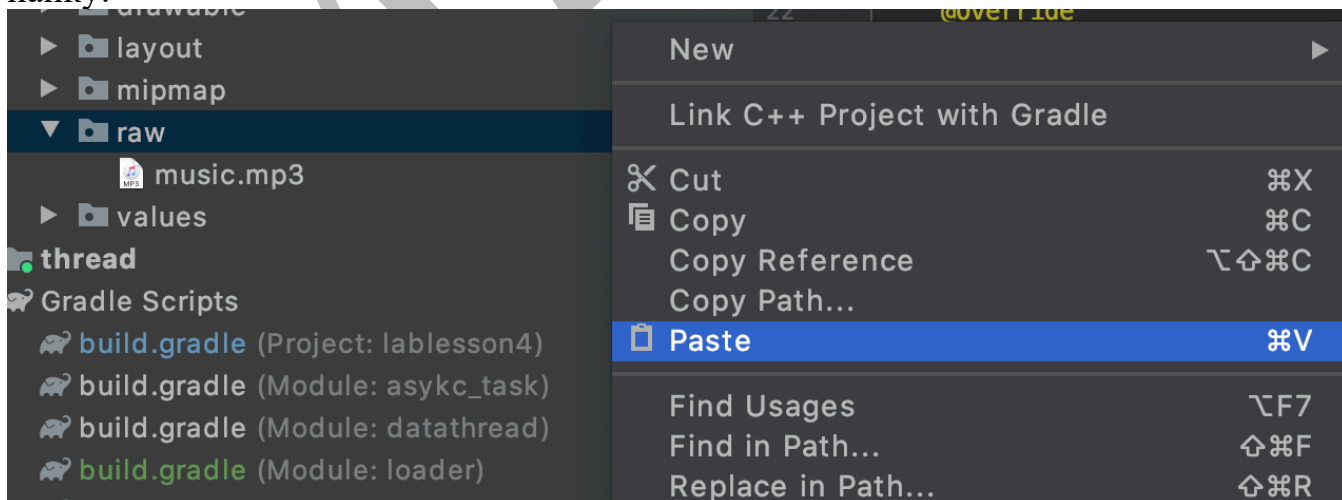


Рисунок 6.2 – Копирование медиа-файла

Далее требуется создать Service, позволяющий проигрывать музыку

в приложении в случаях, когда оно находится в свернутом виде (рисунок.5.3).

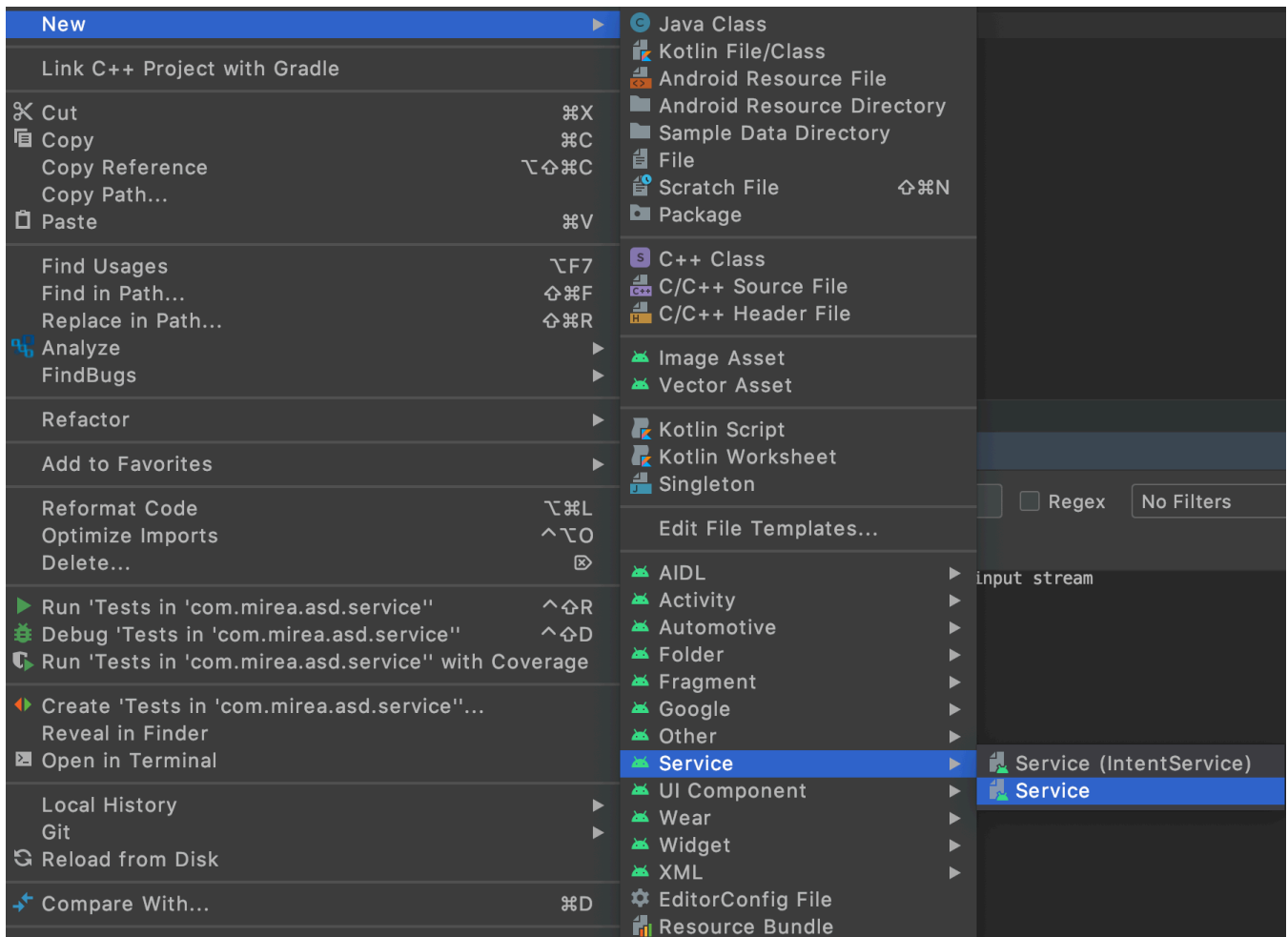


Рисунок 6.3 – Создание Service

Для воспроизведения музыкального файла сервис будет использовать компонент MediaPlayer.

В сервисе переопределяются все четыре метода жизненного цикла:

- метод `onBind()` не имеет реализации;
- в методе `onCreate()` производится инициализация медиа-проигрывателя с помощью музыкального ресурса, который добавлен в папку `res/raw`;
- в методе `onStartCommand()` начинается воспроизведение;
- метод `onDestroy()` завершает воспроизведение.

```

public class PlayerService extends Service {
    private MediaPlayer mediaPlayer;

    @Override
    public IBinder onBind(Intent intent) {
        // TODO: Return the communication channel to the service.
        throw new UnsupportedOperationException("Not yet implemented");
    }

    @Override
    public void onCreate(){
        mediaPlayer=MediaPlayer.create(this, R.raw.music);
        mediaPlayer.setLooping(true);
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId){
        mediaPlayer.start();
        return START_STICKY;
    }
    @Override
    public void onDestroy() {
        mediaPlayer.stop();
    }
}

```

Чтобы управлять сервисом, требуется добавить в layout 2 кнопки для управления сервисом:

```

public void onClickPlayMusic(View view) {
    startService(
        new Intent(MainActivity.this, PlayerService.class));
}

public void onClickStopMusic(View view) {
    stopService(
        new Intent(MainActivity.this, PlayerService.class));
}

```

Для запуска сервиса в классе Activity определен метод startService(), в который передается объект Intent. Этот метод посылает команду сервису и производит вызов его метода onStartCommand(), а также указывает системе, что сервис должен продолжать работать до тех пор, пока не будет вызван метод stopService().

Метод stopService() также определен в классе Activity и принимает объект Intent. Он останавливает работу сервиса, вызывая его метод onDestroy().

Регистрация сервиса производится в узле application с помощью добавления элемента <service>. В нем определяется атрибут android:name, который хранит название класса сервиса. И кроме того может принимать еще ряд атрибутов:

- android:enabled: если имеет значение «true», то сервис может создаваться системой. Значение по умолчанию – «true»;
- android:exported: указывает, могут ли другие компоненты приложения

обращаться к сервису;

- `android:icon`: значок сервиса, представляет собой ссылку на ресурс `drawable`;

- `android:isolatedProcess`: если имеет значение `true`, то сервис может быть запущен как специальный процесс, изолированный от остальной системы;

- `android:label`: название сервиса, которое отображается пользователю;

- `android:permission`: набор разрешений, которые должно применять приложение для запуска сервиса;

- `android:process`: название процесса, в котором запущен сервис. Как правило, имеет то же название, что и пакет приложения.