МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Практическое занятие № 4/2ч.

Разработка мобильных приложений

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень: бакалавриат
(бакалавриат, магистратура, специалитет)

Форма обучения: очная
(очная, очно-заочная, заочная)

Направление(-я) подготовки: 09.03.02 «Информационные системы и технологии»
(код(-ы) и наименование(-я))

Институт: кибербезопасности и цифровых технологий
(полное и краткое наименование)

Кафедра: КБ-14 «Цифровые технологии обработки данных»
(полное и краткое наименование кафедры, реализующей дисциплину (модуль))

Используются в данной редакции с учебного года 2024/25
(учебный год цифрами)

Проверено и согласовано «____» _____20___г. _____ _____
(подпись директора Института/Филиала с расшифровкой)

Москва 2025 г.

ОГЛАВЛЕНИЕ

1 ПРИВЯЗКА ГРАФИЧЕСКИХ КОМПОНЕНТОВ

Требуется создать новый проект *ru.mirea.«фамилия».Lesson4.*

Компания Google в 2019 году представила дополнительный способ связывания текста с кодом – «*viewBinding*». «*ViewBinding*» позволяет сокращать объем кода для взаимодействия с «*view*». При активированном «*ViewBinding*» генерируется «*binding*» классы для каждого файла разметки («*layout*»). Объект сгенерированного «*binding*» класса содержит ссылки на все графические компоненты с указанным «*android:id*» из файла разметки.

В файл разметки «*activity_main*» требуется добавить несколько элементов графического интерфейса («*button*», «*textView*» и т.д.) и установить соответсвующий идентификатор.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/an-
droid"
  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"
  tools:context=".MainActivity">

  <EditText
    android:id="@+id/editTextMirea"
    .../>
  <TextView
    android:id="@+id/textViewMirea"
    ... />
  <Button
    android:id="@+id/buttonMirea"
    ... />
  <Button
    ... />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Чтобы включить «*ViewBinding*» в файле сборки модуля «*gradle*» требуется добавить следующую запись:

```
buildFeatures {
    viewBinding = true
}
```

После изменения «*build.gradle*»-файла требуется заново собрать проект. Каждый сгенерированный «*binding*» класс содержит ссылку на корневой «*view*» разметки («*root*») и ссылки на все «*view*», которые имеют «*id*». Имя генерируемого класса содержит в себе название файла разметки, переведенное в соответствие

стилю написания составных слов «*camel case*» + «*Binding*» (слова пишутся слитно без пробелов, при этом каждое слово внутри фразы пишется с прописной буквы).

Путь до сгенерированного класса представлен на рисунке 1.1.

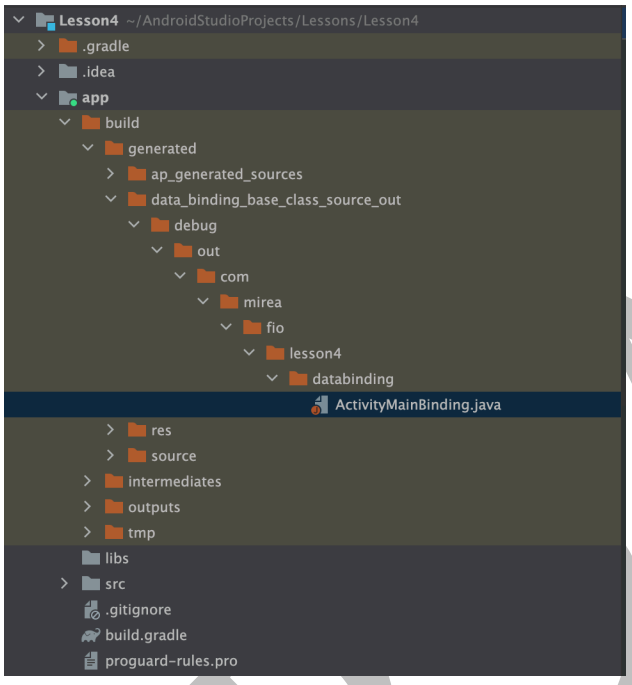


Рисунок 1.1 – Путь размещения сгенерированного файла *«ActivityMainBinding»*

Основная логика класса содержится в методе «*bind»,* в котором производится поиск ссылок и объектов для связывания.

```
public final class ActivityMainBinding implements ViewBinding {
...
@NonNull
public static ActivityMainBinding bind(@NonNull View rootView) {
 int id;
 missingId: {
  id = R.id.buttonMirea;
  Button buttonMirea = ViewBindings.findChildViewById(rootView, id);
  if (buttonMirea == null) {
   break missingId;
  }
  id = R.id.editTextMirea;
  EditText editTextMirea = ViewBindings.findChildViewById(rootView, id);
  if (editTextMirea == null) {
   break missingId;
  }
  id = R.id.textViewMirea;
  TextView textViewMirea = ViewBindings.findChildViewById(rootView, id);
  if (textViewMirea == null) {
   break missingId;
  }
  return new ActivityMainBinding((ConstraintLayout) rootView, buttonMirea, editTextMirea,
      textViewMirea);
 }
 String missingId = rootView.getResources().getResourceName(id);
 throw new NullPointerException("Missing required view with ID: ".concat(missingId));
}
```

Создание объекта класса «*ActivityMainBinding*» осуществляется с помощью вызова статического метода «*inflate*». «*LayoutInflater*» – это класс, который умеет из содержимого «*layout*»-файла создавать «*View*»-элемент. Существует несколько реализаций данного метода с различными параметрами, но результатом их выполнения всегда является «*View*». У экземпляра класса «*ActivityMainBinding*» возможно вызвать метод «*getRoot*», возвращающий корневой элемент «*ConstraintLayout*», и предназначенный для установки «*contentView*» в активности.

```
public class MainActivity extends AppCompatActivity {
    private ActivityMainBinding binding;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());
    }
}
```

Для файла разметки «*activity_main.xml*» будет сгенерирован класс «*ActivityMainBinding*», содержащий 3 поля: «*EditText – editTextMirea; TextView – textViewMirea; Button – buttonMirea».* Для второго элемента «*Button*» не будет сгенерирован идентификатор, так как он не имеет «*android:id*». Далее представлен пример получения доступа к графическим элементам:

```
binding.editTextMirea.setText("Мой номер по списку №___");
binding.buttonMirea.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Log.d(MainActivity.class.getSimpleName(),"onClickListener");
    }
});
```

Основным преимуществом «*View Binding*» является то, что никакие ссылки на объекты не будут иметь нулевых или недействительных значений («*Null safety*») и исключена возможность возникновения ошибок согласования типов во время выполнения (*«Type safety»*). Стоит отметить, если какая-либо «*view*» размещена в одной конфигурации разметки, но отсутствует в другой («*layout-land*», например), то в классе «*binding*» будет сгенерировано «*@Nullable*» поле.

Использование «*View Binding*» позволяет выявлять все несоответствия между разметкой и кодом на этапе компиляции, что приведет к устранению ошибок во время выполнения приложения.

Настройка экземпляра класса привязки во фрагментах производится аналогичным образом. В «*onCreateView*» требуется вызвать метод «*inflate*» в сгенерированном классе связывания и получить ссылку на корневое представление («*getRoot*»). Возвращенный корневой вид из «*onCreateView*» отобразит его на экране.

```
public class GalleryFragment extends Fragment {
  private FragmentGalleryBinding binding;
  public View onCreateView(@NonNull LayoutInflater inflater,
                ViewGroup container, Bundle savedInstanceState) {
    binding = FragmentGalleryBinding.inflate(inflater, container, false);
    View root = binding.getRoot();
    final TextView textView = binding.textGallery;
    return root;
  }

  @Override
  public void onDestroyView() {
    super.onDestroyView();
    binding = null;
  }
}
```

Требуется создать экран музыкального плеера с использованием «*binding*» для горизонтальной и портретной ориентации.

2 ОСНОВНЫЕ ПОНЯТИЯ АСИНХРОННОЙ РАБОТЫ В ОС «ANDROID»

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 операций 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.

2.1 Архитектура ОС «Android»

Центральными для системы являются демон «*zygote*» и запущенные им процессы (рисунок 2.1). 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.

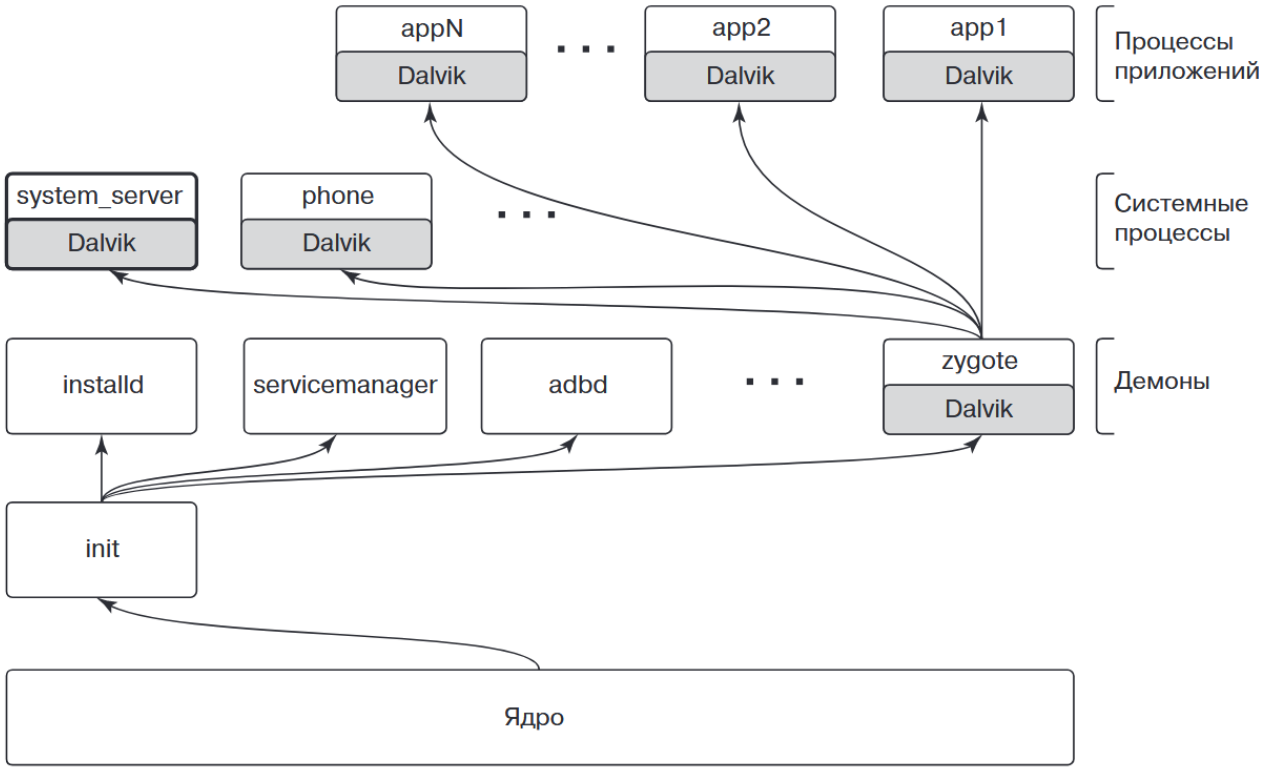


Рисунок 2.1 – Этапы загрузки ОС «*Android*»

В ОС «*Linux*» каждая запущенная программа является отдельным процессом. Каждый процесс обладает уникальным номером и собственным 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. Все новые процессы, использующие среду «*Dalvik*» (системные или прикладные), ответвляются от «*zygote*», что позволяет им начинать выполнение с уже готовой к работе средой. Также «*Zygote*»-процесс осуществляет предварительную загрузку многих частей «*Android*»-среды, которые обычно используются в системе и приложениях, а также загружает ресурсы и другие часто используемые компоненты.

2.2 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 отправляется команда запуска нового процесса. В данной команде содержится описание создаваемой песочницы: UID, под которым должен запуститься новый процесс, и различные ограничения. Таким образом, «*zygote*» запускается 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 2.1).

Таблица 2.1 – Значения UID различного типа процессов в ОС «*Android*»

UID	**Тип процесса**
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

Действие какой-либо команды или командной оболочки инициирующей (порождающей) новый подпроцесс для выполнения какой-либо работы, называется ветвлением («*forking*») процесса. Новый процесс называется «дочерним» («потомком»), а породивший его процесс — «родительским» (или «предком»). В результате и потомок и предок продолжают исполняться одновременно — параллельно друг другу. 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 2.2. 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 и загрузить запускаемый код приложения или системный код.

Рисунок 2.2 – Создание нового процесса

Порядок запуска активности в новом процессе показан на рисунке 2.3 и состоит из следующих этапов:

1. Один из процессов (например, предназначенный для запуска приложений) осуществляет вызов диспетчера активностей с намерением, дающим описание запускаемой активности.

2. Диспетчер активностей отправляет диспетчеру пакетов требование на проверку разрешения 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: «Цель моего запуска?».

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.

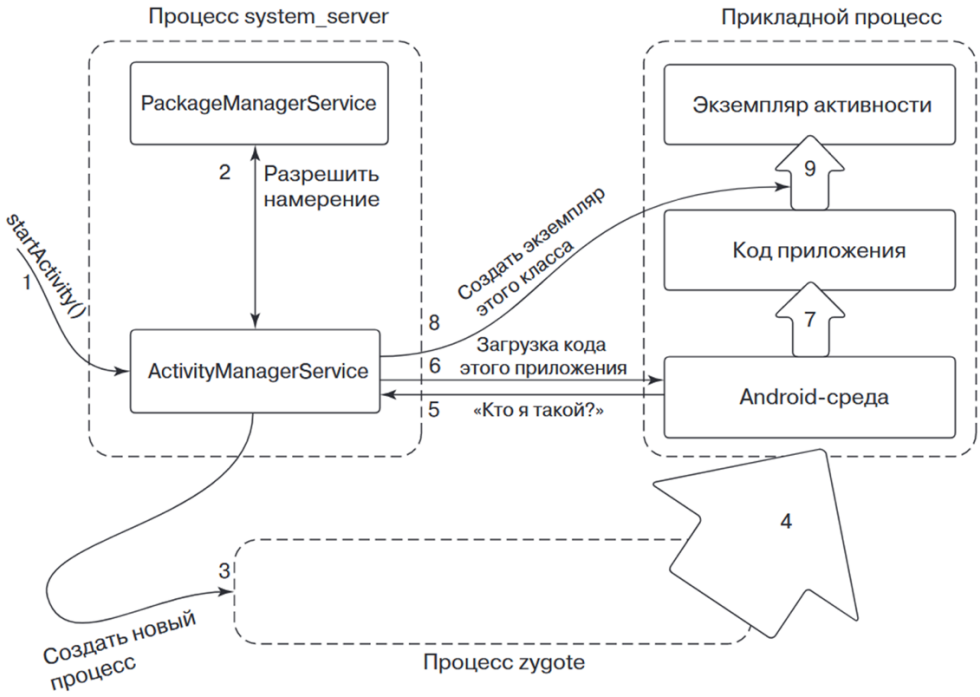


Рисунок 2.3 – Создание нового процесса

По умолчанию компоненты одного приложения работают в одном процессе. Однако, если необходимо контролировать, к какому процессу принадлежит 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 данный 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.

2.3 Жизненный цикл процессов

Операционная 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 ОС «*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 по основным категориям. Основные категории приведены в таблице 2.2. В последнем столбце показано типовое значение «*oom_adj*», которое назначается процессу данного типа.

Таблица 2.2– Основные категории процессов и их значимость

Категория	**Описание**	**oom_adj**
SYSTEM	Системные процессы и демоны	-16
PERSISTENT	Постоянно работающие прикладные процессы	-12
FOREGROUND	Процессы, взаимодействующие в данный момент с пользователем	0
VISIBLE	Процессы, видимые пользователю	1

Категория	Описание	oom_adj
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 так далее. Внутри конкретного уровня «*oom_adj*» механизм сначала уничтожит процессы, которые используют больше оперативной памяти, а затем перейдет к уничтожению тех, которые используют меньше памяти.

2.4 Потоки

Приложения в ОС «*Android*» запускаются в собственном процессе и как минимум имеется один поток. По умолчанию все компоненты одного приложения работают в одном процессе и потоке (называется «главным потоком»). В главном потоке выполняется диспетчеризация событий пользовательского интерфейса пользователя. Он также является потоком, в котором приложение взаимодействует с компонентами из набора инструментов пользовательского интерфейса ОС «*Android*» (компонентами из пакетов «*android.widget*» и «*android.view*»). Если в приложении имеются какие-либо визуальные элементы, то в данном потоке запускается объект класса «*Activity*», отвечающий за отображение на дисплее интерфейса («*user interface*», UI).

В главном потоке «*Activity*» должно быть, как можно меньше вычислений, т.к. единственная его задача – отображать UI. В случаях, когда главный поток будет занят вычислением сложных операций, то он не сможет обрабатывать пользовательские события. Если ожидание продолжается больше нескольких секунд, ОС «*Android*» отобразит сообщение ANR (от англ. «*application not responding*» – приложение не отвечает) с предложением принудительно завершить приложение (рисунок 2.4).

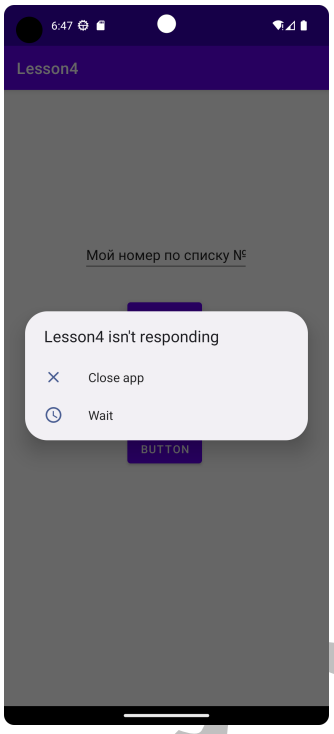


Рисунок 2.4 – Сообщение «*application not responding*»

Для того, чтобы получить данное сообщение достаточно в главном потоке начать работу с файловой системой, сетью, криптографией и так далее.

2.5 Задание

Создать новый модуль. В меню «*File* | *New* | *New Module* | *Phone & Tablet Module* | *Empty Views Activity*». Имя модуля «thread».

На экране требуется разместить элементы «*Button*» и «*TextView*». Инициализацию графических компонентов осуществить с помощью «*Binding*».

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 возможно 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*» – получение имени потока;
- «*getPriority*» – получение приоритета потока;
- «*isAlive*» – определение состояния потока;

– «*join*» – запуск потока с момента завершения другого потока;

– «*run*» – реализация основной задачи;

– «*sleep*» – приостановка поток на заданное время;

– «*getStackTrace*» – получение стэка вызываемых методов;

– «*start*» – запуск потока.

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("Имя текущего потока: " + mainThread.getName());
// Меняем имя и выводим в текстовом поле
mainThread.setName("МОЙ НОМЕР ГРУППЫ: XX, НОМЕР ПО СПИСКУ: XX, МОЙ ЛЮБИИМЫЙ ФИЛЬМ: XX");
infoTextView.append("\n Новое имя потока: " + mainThread.getName());
Log.d(MainActivity.class.getSimpleName(), "Stack: " + Arrays.toString(mainThread.getStackTrace()));
```

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

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

```
binding.buttonMirea.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
    long endTime = System.currentTimeMillis() + 20 * 1000;

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

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

В ОС «*Linux*» (в том числе и в ОС «Android») каждый поток исполнения имеет собственный приоритет, значение которого варьируется в пределах от –20 до 19, где меньшее число означает более высокий приоритет (рисунок 2.5).

```
-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                                 ANDROID_PRIORITY_URGENT_DISPLAY + 2    -6
10                 MAX_PRIORITY   ANDROID_PRIORITY_URGENT_DISPLAY        -8
```

Рисунок 2.5 – Шкала приоритета потоков

Ядро ОС «*Linux*» объединяет потоки в так называемые контрольные группы («*groups*») в зависимости от разных условий, таких, например, как видимость приложения на экране. Таким способом возможно получить значения потоков в группе и их приоритет:

```
Log.d(MainActivity.class.getSimpleName(), "Group: " + mainThread.getThreadGroup());
```

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

– *«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.

Базовым классом для потоков в ОС «*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();
```

Требуется перенести функционал, замедляющий работу приложения, в отдельный поток. Для этого создаётся экземпляр класса «*Runnable*», у которого имеется метод «*run*». Далее создаётся объект «*Thread*», в конструкторе у которого указывается созданный «*Runnable*». После этого возможно запускать новый поток с помощью метода «*start*».

```
public class MainActivity extends AppCompatActivity {
  private ActivityMainBinding binding;
  private int counter = 0;
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ...
    binding.buttonMirea.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
        new Thread(new Runnable() {
          public void run() {
            int numberThread = counter++;
            Log.d("ThreadProject", String.format("Запущен поток № %d студентом группы № %s номер по
списку № %d ", numberThread, "БСБО-XX-XX", -1));
            long endTime = System.currentTimeMillis() + 20 * 1000;
            while (System.currentTimeMillis() < endTime) {
              synchronized (this) {
                try {
                  wait(endTime - System.currentTimeMillis());
                  Log.d(MainActivity.class.getSimpleName(), "Endtime: " + endTime);
                } catch (Exception e) {
                  throw new RuntimeException(e);
                }
              }
              Log.d("ThreadProject", "Выполнен поток № " + numberThread);
            }
          }
        }).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.d*». При каждом нажатии создаётся новый поток, в котором выполняется код (рисунок 2.6).

```
D  Запущен поток № 1 студентом группы № БСБО-XX-XX номер по списку № -1
D  Запущен поток № 2 студентом группы № БСБО-XX-XX номер по списку № -1
D  Запущен поток № 3 студентом группы № БСБО-XX-XX номер по списку № -1
I  Compiler allocated 4533KB to compile void android.view.ViewRootImpl.performTraversals()
D  Запущен поток № 4 студентом группы № БСБО-XX-XX номер по списку № -1
D  Запущен поток № 5 студентом группы № БСБО-XX-XX номер по списку № -1
```

Рисунок 2.6 – Вывод сообщений инструмента Logcat

При данной реализации потоков сложно использовать полноценные возможности дополнительных потоков – отсутствуют возможности изменения задачи и просмотра результатов вычислений.

3 ПЕРЕДАЧА ДАННЫХ МЕЖДУ ПОТОКАМИ

3.1 Задание. Thread в UI

Создать новый модуль. В меню «*File | New | New Module | Phone & Tablet Module | Empty Views Activity*». Имя модуля: «*data_thread*».

Основной поток также называют UI-потоком. 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 {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ....
        final Runnable runn1 = new Runnable() {
            public void run() {
               binding.tvInfo.setText("runn1");
            }
        };
        final Runnable runn2 = new Runnable() {
            public void run() {
                binding.tvInfo.setText("runn2");
            }
        };
        final Runnable runn3 = new Runnable() {
            public void run() {
                binding.tvInfo.setText("runn3");
            }
        };
        Thread t = new Thread(new Runnable() {
            public void run() {
                try {
                    TimeUnit.SECONDS.sleep(2);
                    runOnUiThread(runn1);
                    TimeUnit.SECONDS.sleep(1);
                    binding.tvInfo.postDelayed(runn3, 2000);
                    binding.tvInfo.post(runn2);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        t.start();
    }
}
```

Требуется определить в какой последовательности происходит запуск процессов. Изучите методы «*runOnUiThread*», «*postDelayed*», «*post*». В «*TextViwe*» описать в чём различия между элементами и последовательность запуска. У элемента «*TextView*» имеется возможность установки значений строк:

android:maxLines="10"

android:lines="10"

3.2 Задание. Looper

Создать новый модуль. В меню «*File | New | New Module | Phone & Tablet Module | Empty Views Activity*». Имя модуля: «*looper*».

В main_activity.xml требуется добавить «button» и реализовать обработку нажатия.

Одним из способов передачи данных из одного потока в другой является 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).

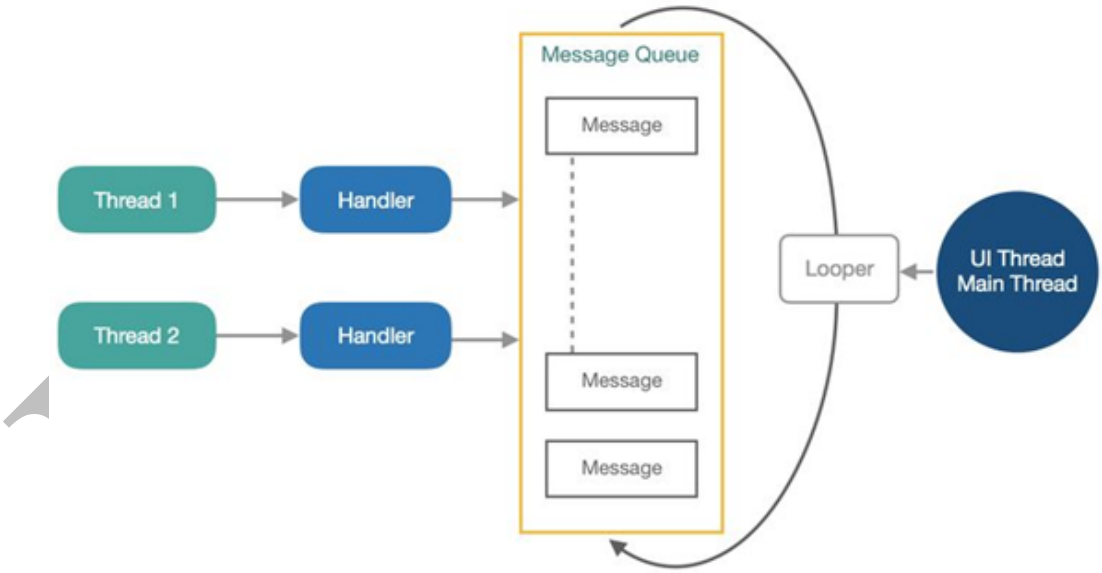


Рисунок 3.1 – Механизм функционирования Looper

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 | Java Class | Ввести имя класса*» (рисунок 3.2).

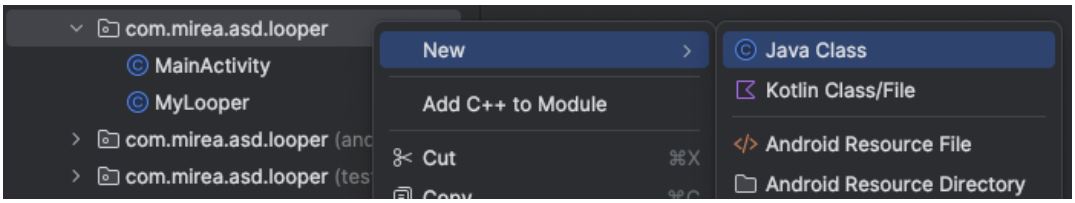


Рисунок 3.2 - Создание нового класса

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

```
public class MyLooper extends Thread{
    public Handler mHandler;
    private Handler mainHandler;
    public MyLooper(Handler mainThreadHandler) {
        mainHandler =mainThreadHandler;
    }

    public void run() {
        Log.d("MyLooper", "run");
        Looper.prepare();                                   Создание цикла (1)
        mHandler = new Handler(Looper.myLooper()) {
            public void handleMessage(Message msg) {
                String data = msg.getData().getString("KEY");
                Log.d("MyLooper get message: ", data);

                int count = data.length();
                Message message = new Message();
                Bundle bundle = new Bundle();
                bundle.putString("result", String.format("The number of letters in the word %s is %d ",data,count));
                message.setData(bundle);
                // Send the message back to main thread message queue use main thread message Handler.
                mainHandler.sendMessage(message);
            }
        };                                                  Получение и отправка данных (2)
        Looper.loop();                                      Запуск цикла (3)
    }
}
```

Запуск потока осуществляется с создания нового объекта «*MyLooper*» и вызова метода «*start*» в методе «*onCreate*». 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.

```
public class MainActivity extends AppCompatActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ActivityMainBinding binding = ActivityMainBinding.inflate(getLayoutInflater());
    setContentView(binding.getRoot());

    Handler mainThreadHandler = new Handler(Looper.getMainLooper()) {
      @Override
      public void handleMessage(Message msg) {
        Log.d(MainActivity.class.getSimpleName(), "Task execute. This is result: " + msg.getData().getString("re-
sult"));
      }
    };
    MyLooper myLooper = new MyLooper(mainThreadHandler);
    myLooper.start();

    binding.editTextMirea.setText("Мой номер по списку №___");
    binding.buttonMirea.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
      ....
    });
  }
}
```

Создание обработчика данных (1)

Создание и запуск потока (2)

Следующий код формирует сообщение, содержащее послание, и производит отправку в другой поток по нажатию на кнопку. В данном примере создается объект класса Message, в который передаются объект Bundle с передаваемыми значениями. Данное сообщение передаётся в «*MessageQueue*» через метод «*sendMessage*».

```
    @Override
    public void onClick(View v) {
        Message msg = Message.obtain();
        Bundle bundle = new Bundle();
        bundle.putString("KEY", "mirea");
        msg.setData(bundle);
        myLooper.mHandler.sendMessage(msg);
    }
```

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

3.3 Задание. Loader

Создать новый модуль. В меню «*File | New | New Module | Phone & Tablet Module | Empty Views Activity*». Имя модуля: «*CryptoLoader*».

Основной задачей большинства мобильных приложений является быстрая и незаметная для пользователя загрузка данных из сети или файловой системы, с последующим отображением их на дисплее. Загрузчики появились

в ОС «*Android 3.0*», упрощая асинхронную загрузку данных. 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 данных;
- автоматически переподключаются к последнему курсору загрузчика при изменении конфигурации. Таким образом, им не требуется повторно запрашивать свои данные.

«*Loader*» – это компонент ОС «*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 тому потребителю. Каждая активность и каждый фрагмент имеет один экземпляр менеджера «*LoaderManager*», который работает с загрузчиками через методы «*initLoader», «restartLoader», «destroyLoader»* (рисунок 3.3). Данный менеджер используется активностью для передачи сообщений о своём уничтожении, а также для того, чтобы «*LoaderManager»* закрыл загрузчики для экономии ресурсов.

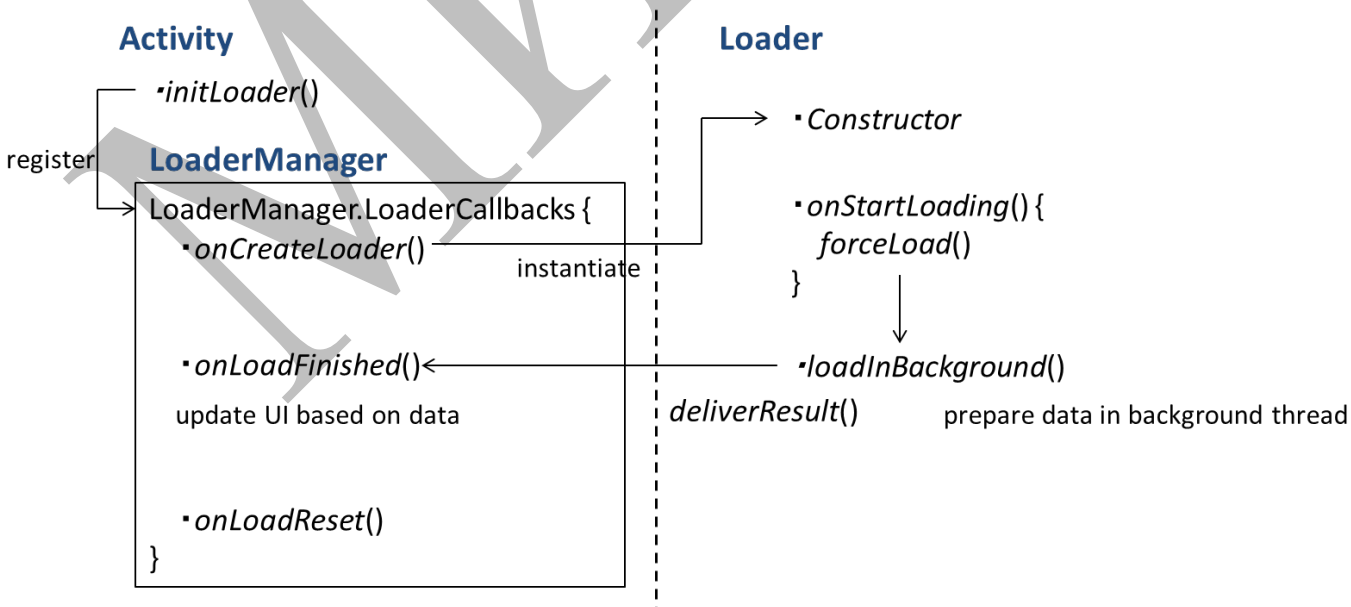


Рисунок 3.3 – Отношение между «*Activity*» и «*LoaderManager*»

«*LoaderManager*» не обладает знаниями, каким образом данные загружаются

в приложение, а лишь выдаёт указания загрузчику начать, остановить, обновить загрузку данных и другие команды. В таблице 3.1 представлена информация об API – интерфейса загрузчика.

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 имеется возможность создания собственных загрузчиков 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 данного загрузчика является лучшим способом асинхронной загрузки данных из «*ContentProvider*» вместо выполнения управляемого запроса через платформу или API-интерфейсы операции.

Приведенные в таблице 3.1 классы и интерфейсы являются наиболее важными компонентами, с помощью которых в приложении реализуется «загрузчик». При создании каждого загрузчика не требуется 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*» требует переопределения трех методов 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) {

    }
}
```

Основные методы обратного вызова интерфейсов:

– 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. В случае его отсутствия, вызывается метод «*onCreateLoader*», где и создается 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 перестать использовать старые данные (поскольку они скоро

будут заменены). 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. Пример реализации представлен на рисунке 3.4.

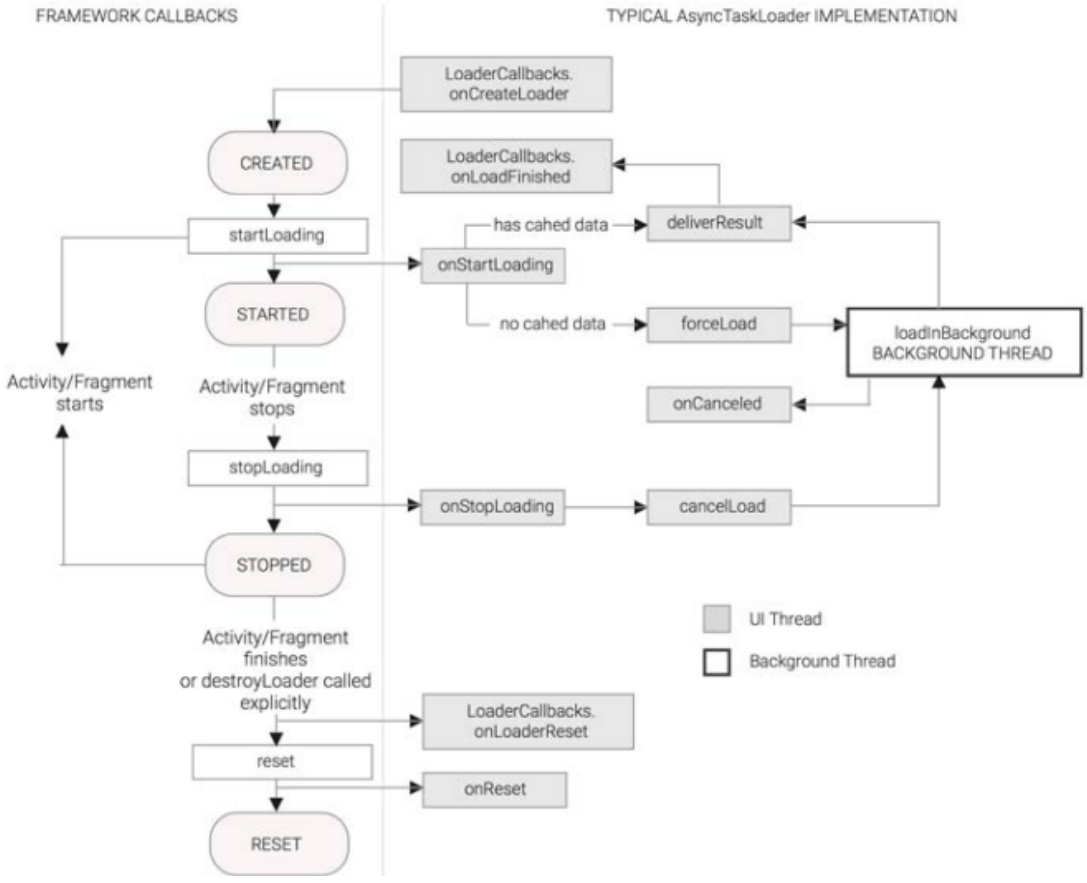


Рисунок 3.4 – Механизм «*AsyncTaskLoader»*

Слушатель получает информацию от загрузчика. Для этого менеджер

регистрирует слушатель «*OnLoadCompleteListener*», который прослушивает события. Когда загрузка данных закончилась, вызывается метод «*onLoadFinished*». 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. Основной задачей будет являться освобождение памяти.

Для реализации собственного загрузчика, требуется создание класса и переопределение методов «*loadInBackground*» и «*onStartLoading*». Имя класса *MyLoader* («*New->JavaClass->…*»):

```
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 качестве примера, в методе «*loadInBackground*» осуществляется загрузка данных строкового типа. Возможно размещение запроса к базе данных, сетевой части и т.д. В данном случае возвращаются данные, которые указаны при создании объекта через 5 секунд. 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 final int LoaderID = 1234;
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ……
  }

  public void onClickButton(View view) {
    Bundle bundle = new Bundle();
    bundle.putString(MyLoader.ARG_WORD, "mirea");
    LoaderManager.getInstance(this).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);
    }
    throw new InvalidParameterException("Invalid loader id");
  }

  @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.

Задание! В созданном модуле требуется добавить элементы «*EditText*» и «*Button*». Пользователь вводит фразу в «*EditText*», далее она шифруется с помощью алгоритма AES и передается вместе с ключом в «*Loader*». В «*Loader*» происходит дешифровка фразы и последующая передача в «*MainActivity*». Дешифрованная фраза отображается с помощью «*toast*» или «*snackBar*».

В задании используется симметричный алгоритм шифрования. Для создания ключа шифрования далее приведен пример программной реализации:

```
public static SecretKey generateKey(){
   try {
      SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
      sr.setSeed("any data used as random seed".getBytes());
      KeyGenerator kg = KeyGenerator.getInstance("AES");
      kg.init(256, sr);
      return new SecretKeySpec((kg.generateKey()).getEncoded(), "AES");
   } catch (NoSuchAlgorithmException e) {
      throw new RuntimeException(e);
   }
}
```

После формирования ключа шифрования требуется получить значение введенного текста и передать их в метод шифрования:

```
public static byte[] encryptMsg(String message, SecretKey secret) {
   Cipher cipher = null;
   try {
      cipher = Cipher.getInstance("AES");
      cipher.init(Cipher.ENCRYPT_MODE, secret);
      return cipher.doFinal(message.getBytes());
   } catch (NoSuchAlgorithmException | NoSuchPaddingException | InvalidKeyException |
         BadPaddingException | IllegalBlockSizeException e) {
      throw new RuntimeException(e);
   }
}
```

Передачу ключа и текста в «*Loader*» возможно оуществить с помощью «*Bundle*».

```
// Отправка данных в Loader
Bundle bundle = new Bundle();
bundle.putByteArray(MyLoader.ARG_WORD, shiper);
bundle.putByteArray("key", key.getEncoded());
LoaderManager.getInstance(this).initLoader(LoaderID, bundle, this);
---------------------------------------------------------------

// Обработка данных в Loader
byte[] cryptText = args.getByteArray(ARG_WORD);
byte[] key = args.getByteArray("key");
// Восстановление ключав
SecretKey originalKey = new SecretKeySpec(key, 0, key.length, "AES");
```

Восстановленный ключ и зашифрованный текст требуется передать в метод дешифрования.

```
public static String decryptMsg(byte[] cipherText, SecretKey secret){
   /* Decrypt the message */
   try {
      Cipher cipher = Cipher.getInstance("AES");
      cipher.init(Cipher.DECRYPT_MODE, secret);
      return new String(cipher.doFinal(cipherText));
   } catch (NoSuchAlgorithmException | NoSuchPaddingException| IllegalBlockSizeException
         | BadPaddingException | InvalidKeyException e) {
      throw new RuntimeException(e);
   }
}
```

Внимание!

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

4 СЕРВИС

Сервисы являются компонентом приложения, который может выполнять длительные операции в фоновом режиме и не содержит пользовательского интерфейса за исключением возможности создания уведомлений. Компонент приложения может запустить службу, которая продолжит работу в фоновом режиме даже в том случае, когда пользователь перейдет в другое приложение, а также может привязаться к службе для взаимодействия с ней и даже выполнять межпроцессное взаимодействие (от англ. «*Inter-process communication»,* IPC). Например, служба может обрабатывать сетевые транзакции, воспроизводить музыку, выполнять ввод-вывод файла или взаимодействовать с поставщиком контента, и все это в фоновом режиме. Службы бывают следующих видов:

– переднего плана, выполняющие заметные для пользователя операции. Например, аудиоприложения используют службу переднего плана для воспроизведения звукозаписей. Услуги переднего плана должны отображать уведомление о состоянии сервиса и продолжают выполняться, даже если пользователь не взаимодействует с приложением;

– фоновая служба выполняет операцию, которая не отображается напрямую пользователю. Например, если приложение выполняет задачу сжатия данных в хранилище приложения;

– 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 в том числе между различными 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.

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.

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 имеют следующие методы жизненного цикла:

- «*onCreate*» вызывается при создании сервиса;
- «*onStartCommand*» вызывается при получении сервисом команды, отправленной с помощью метода «*startService*»;
- «*onBind*» вызывается при закреплении клиента за сервисом с помощью метода «*bindService*»;
- «*onDestroy*» вызывается при завершении работы сервиса.

4.1 Создание Service

Создать новый модуль. В меню «*File | New | New Module | Phone & Tablet Module | Empty Views Activity*». Имя модуля: «*ServiceApp*».

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):

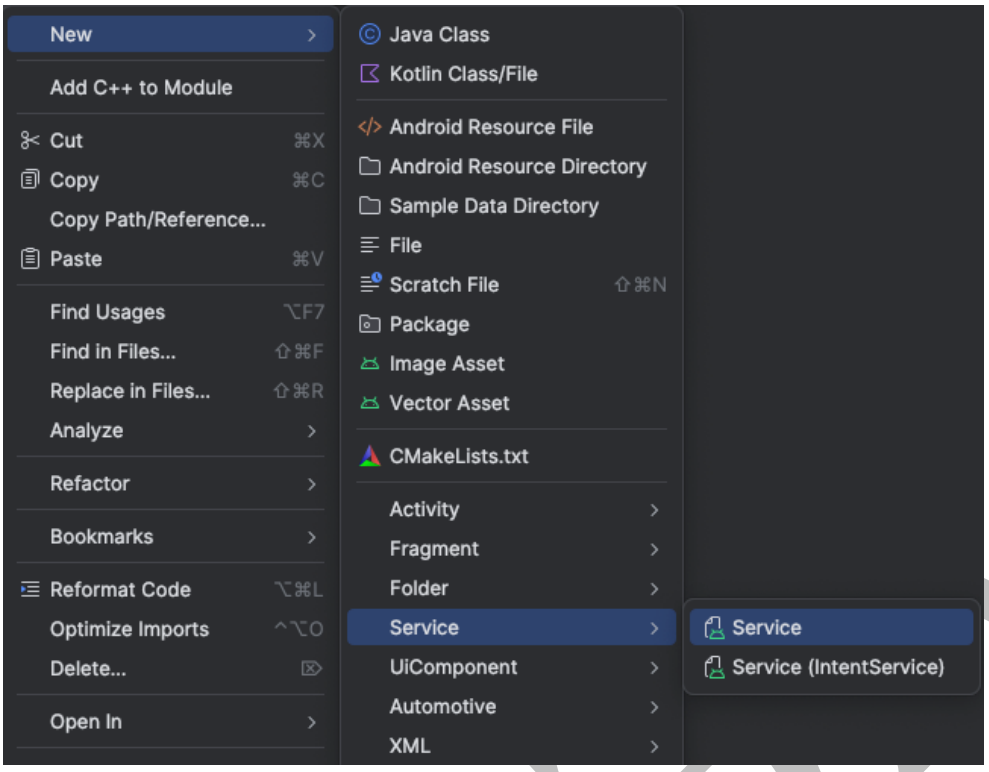


Рисунок 4.1 – Создание Service

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. Требуется установить имя сервиса «PlayerService».

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

Т.к. сервис является компонентом приложения, требуется убедиться в существовании записи о сервисе в секции «*application*» манифест-файла.

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

4.2 Жизненный цикл сервиса

Подобно активностям служба имеет свои методы жизненного цикла:

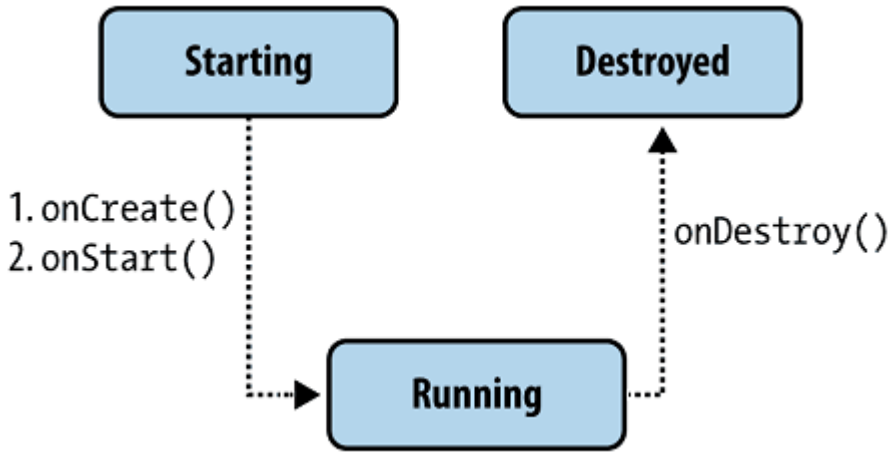


Рисунок 4.2 – Жизненный цикл сервиса

Для быстрого создания заготовок методов родительских классов с целью их переопределения 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 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 жизнь службы начинается с вызова метода «*onStartCommand*». Этому методу передается объект «*Intent*», который передавался в метод «*startService*».

Службу возможно запустить вызовом метода «*Context.startService»*, а 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 данное 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*»). Примером таких служб являются службы, 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 получат 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.

Требуется добавить функционал воспроизведения аудиофайла. В первую очередь необходимо добавить медиа файл (типа «.mp3» и т.д.) в ресурсы. В активности добавить две кнопки «*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 директории «*res/raw*», поэтому следует создать требуемую папку в проекте. Вызвать контекстное меню у папки «*res*|*New*|*Android Resource Directory*» (рисунок.4.3).

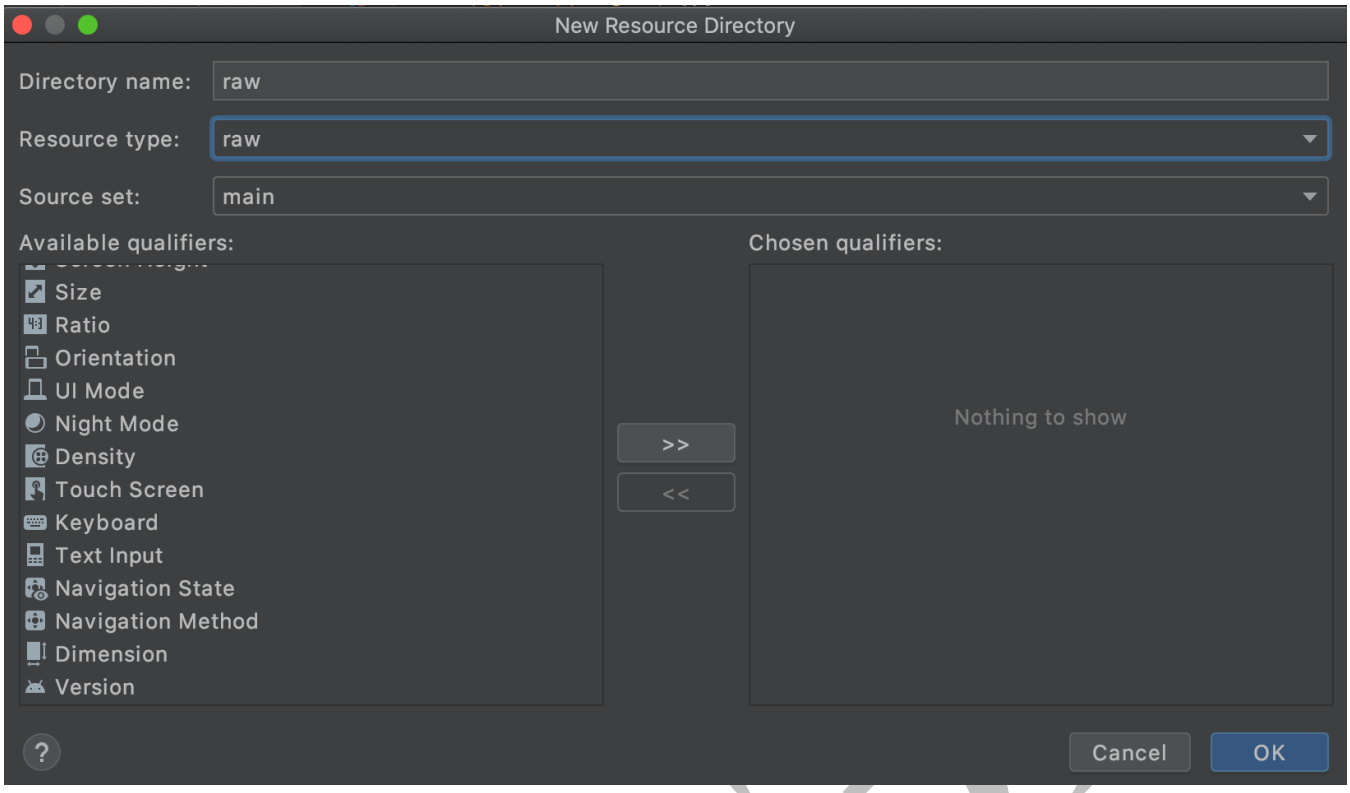


Рисунок 4.3 – Создание папки для хранения медиафайлов

После этого требуется скачать файл «.mp3» и скопировать в данную папку (рисунок 4.4).

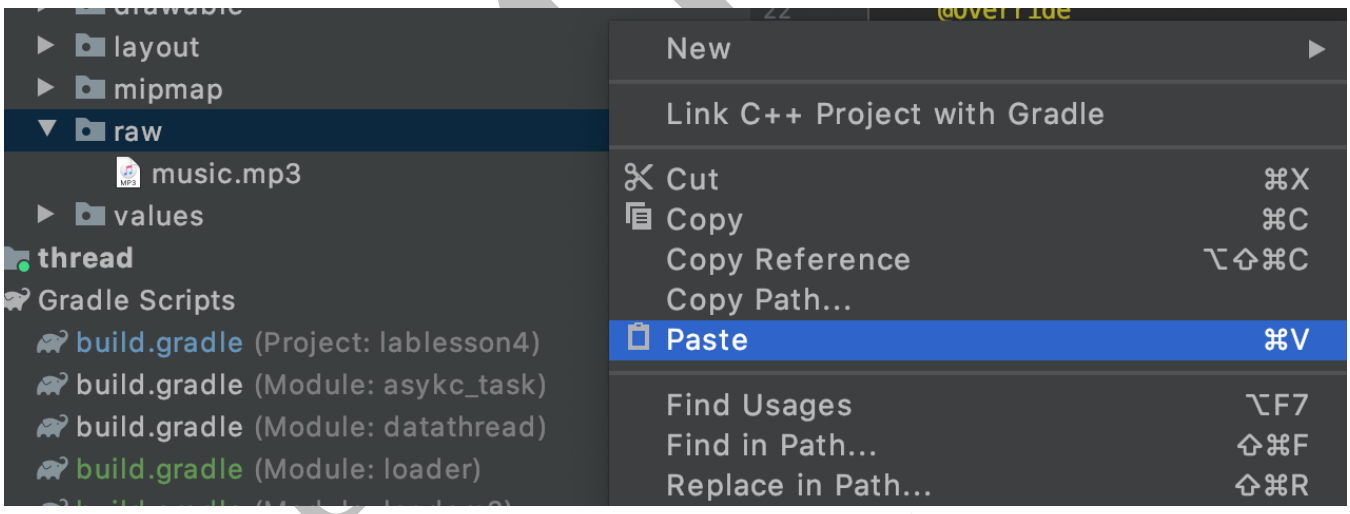


Рисунок 4.4 – Копирование медиафайла

Далее требуется создать компонент «сервис», предназначенный для проигрывания музыки в приложении в случаях, когда оно находится в свернутом виде. Для воспроизведения музыкального файла в сервисе будет использоваться компонент «*MediaPlayer*».

В сервисе переопределяются все четыре метода жизненного цикла:

– метод «*onBind*» не имеет реализации;

– в методе «*onCreate*» производится инициализация медиапроигрывателя 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;
  public static final String CHANNEL_ID = "ForegroundServiceChannel";
  @Override
  public IBinder onBind(Intent intent) {
     throw new UnsupportedOperationException("Not yet implemented");
  }
  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    mediaPlayer.start();
    mediaPlayer.setOnCompletionListener(new MediaPlayer.OnCompletionListener() {
      public void onCompletion(MediaPlayer mp) {
        stopForeground(true);
      }
    });
    return super.onStartCommand(intent, flags, startId);
  }
  @Override
  public void onCreate() {
    super.onCreate();
    NotificationCompat.Builder builder = new NotificationCompat.Builder(this, CHANNEL_ID)
        .setContentText("Playing....")
        .setSmallIcon(R.mipmap.ic_launcher)
        .setPriority(NotificationCompat.PRIORITY_HIGH)
        .setStyle(new NotificationCompat.BigTextStyle()
            .bigText("best player..."))
        .setContentTitle("Music Player");
    int importance = NotificationManager.IMPORTANCE_DEFAULT;
    NotificationChannel channel = new NotificationChannel(CHANNEL_ID, "Student FIO Notification", im-
portance);
    channel.setDescription("MIREA Channel");
    NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
    notificationManager.createNotificationChannel(channel);
    startForeground(1, builder.build());

    mediaPlayer= MediaPlayer.create(this, R.raw.music);
    mediaPlayer.setLooping(false);

  }
  @Override
  public void onDestroy() {
    stopForeground(true);
    mediaPlayer.stop();
  }
}
```

При воспроизведении композиции реализовано оповещение пользователя о проигрываемой композиции, а с помощью интерфейс обратного вызова «*MediaPlayer.OnCompletionListener*» отслеживается окончание воспроизведения аудиозаписи и удаление уведомления. В уведомлении требуется указать СВОЕ

название композиции (ИНАЧЕ РАБОТА НЕ БУДЕТ ПРИНЯТА).

Для создания уведомлений из фонового сервиса требуется набор разрешений:

```
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
<uses-permission android:name="android.permission.POST_NOTIFICATIONS" />
```

Регистрация сервиса производится в узле «*application*» с помощью добавления тэга «*service*». В нем определяется атрибут «*android:name*», который хранит название класса сервиса. Также имеется возможность установки дополнительных атрибутов:

- «*android:enabled*»: если 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 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.

Далее требуется в классе «*MainActivity*» добавить проверку на получение разрешений и обработчик нажатий на кнопки управления сервисом:

```
public class MainActivity extends AppCompatActivity {
    ActivityMainBinding binding;
    private int PermissionCode = 200;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        View view = binding.getRoot();
        setContentView(view);
        if (ContextCompat.checkSelfPermission(this, POST_NOTIFICATIONS) == PackageManager.PERMIS-
SION_GRANTED) {
            Log.d(MainActivity.class.getSimpleName().toString(), "Разрешения получены");
        } else {
            Log.d(MainActivity.class.getSimpleName().toString(), "Нет разрешений!");
            ActivityCompat.requestPermissions(this, new String[]{POST_NOTIFICATIONS, FOREGROUND_SERVICE}, Per-
missionCode);
        }

        binding.button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent serviceIntent = new Intent(MainActivity.this, PlayerService.class);
                ContextCompat.startForegroundService(MainActivity.this, serviceIntent);
            }
        });

        binding.button2.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                stopService(
                        new Intent(MainActivity.this, PlayerService.class));
            }
        });
    }
}
```

Для запуска сервиса в классе «*Activity*» используется метод «*startForegroundService*», в который передается объект «*Intent*». Этот метод посылает команду сервису и производит вызов его метода «*onStartCommand*». Вызов метода «*stopService*» в классе Activity останавливает работу сервиса, вызывая его метод «*onDestroy*».

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. Типовые задачи, 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 большого объема файлов.

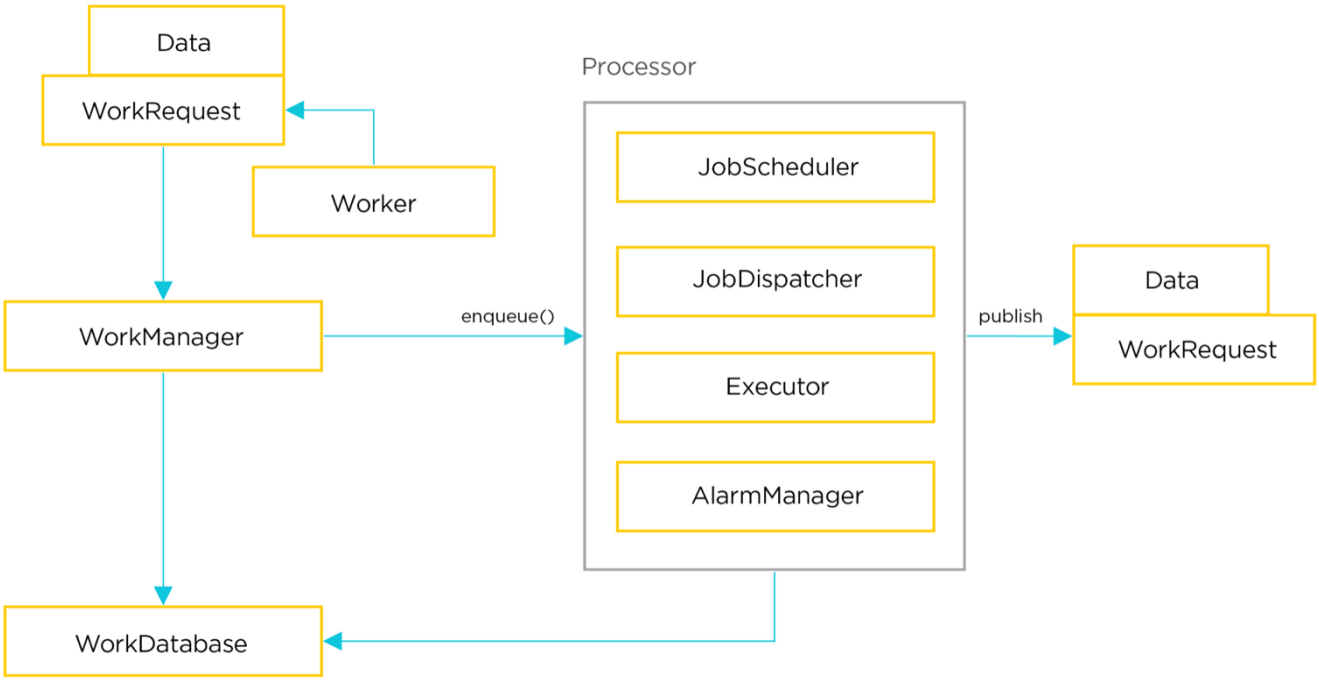
С каждым выходом новой версии ОС «*Android*», разработчики операционной системы всё больше и больше уделяли внимание оптимизации времени работы батареи телефона. Начиная с Android 6.0 Marshmallow разработчики Google предложили так называемый «*Doze mode*» – режим активируется при условиях если, устройство находится без движения и без зарядки в течении часа и почти все приложения перестают выполнять какую-либо фоновую работу и потреблять энергию батареи. 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 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, «*WorkManager»* имеет удобный интерфейс, позволяя разработчику сконцентрироваться на бизнес-логике задачи. Основные классы механизма «*WorkManager»*:

– «*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, необходимый 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 новый модуль. В меню «*File | New | New Module | Phone & Tablet Module | Empty Views Activity*». Имя модуля: «*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:2.10.0")
    // ...
}
```

В модуле требуется 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);
    }
}
```

Далее приведен пример установки требований наличия интернета без тарификации и нахождения устройства на зарядке для запуска задачи.

```
Constraints constraints = new Constraints.Builder()
    .setRequiredNetworkType(NetworkType.UNMETERED)
    .setRequiresCharging(true)
    .build();

WorkRequest uploadWorkRequest =
    new OneTimeWorkRequest.Builder(UploadWorker.class)
        .setConstraints(constraints)
        .build();
```

6 КОНТРОЛЬНОЕ ЗАДАНИЕ

В проекте «*MireaProject*» создать отдельный фрагмент выполнения фоновой задачи и реализовать её выполнение с помощью применения механизма «*Worker*», либо одной из разновидностей сервисов.