Введение

С приходом многоядерности появилось понятие ***параллелизма*** — одновременного выполнения нескольких вычислений. До этого все задачи процессор выполнял последовательно. Конечно, и при одноядерных процессорах можно было просматривать сайты в браузере, слушать музыкальный плеер и в это время копировать файлы на флешку, для пользователя все это выглядело как одновременное (параллельное) выполнение, но на самом деле над этими процессами работал всего один процессор. Для того чтобы создать видимость параллельного выполнения, процессы (программы) разделялись на потоки. Процессор поочередно выполнял потоки из разных процессов, таким образом создавалось впечатление параллелизма, это называется — псевдопараллелизмом.

Что же такое поток? ***Поток (Thread)*** можно представить как последовательность команд программы, которая претендует на использование процессора вычислительной системы для своего выполнения. Потоки одной и той же программы работают в общем адресном пространстве и, тем самым, разделяют (совместно используют) данные программы.

Разработка многопоточных программ — непростая задача. При планировании многопоточной программы оптимальным вариантом является разделение последовательности вычислений на потоки, число которых равно количеству процессоров. Если сделать много потоков, то это не ускорит выполнение программы, а затормозит ее, потому что переключение между потоками требует некоторого дополнительного времени.

В Android приложениях, на языке Java, мы можем использовать несколько инструментов для работы с потоками. Давайте их рассмотрим.

Часть 1. Класс Thread. Интерфейс Runnable.

Тот поток, с которого начинается выполнение программы, называется главным. В языке Java, после создания процесса, выполнение главного потока начинается с метода **main().** Затем, по мере необходимости, в заданных программистом местах, и при выполнении заданных им же условий, запускаются другие, побочные потоки.

Существует универсальный способ создания потоков (**Threads**). Он применим не только в Android**,** но и в любых Java-приложениях. Для этого в Java предусмотрены класс **Thread**, или интерфейс **Runnable**. Предлагается два варианта создания параллельной программы. Можно создать объект, наследуемый от класса **Thread**, переопределив в нем метод **run(),** или можно создать объекты, реализующие интерфейс **Runnable**.

Класс **Thread** содержит несколько методов для управления потоками.

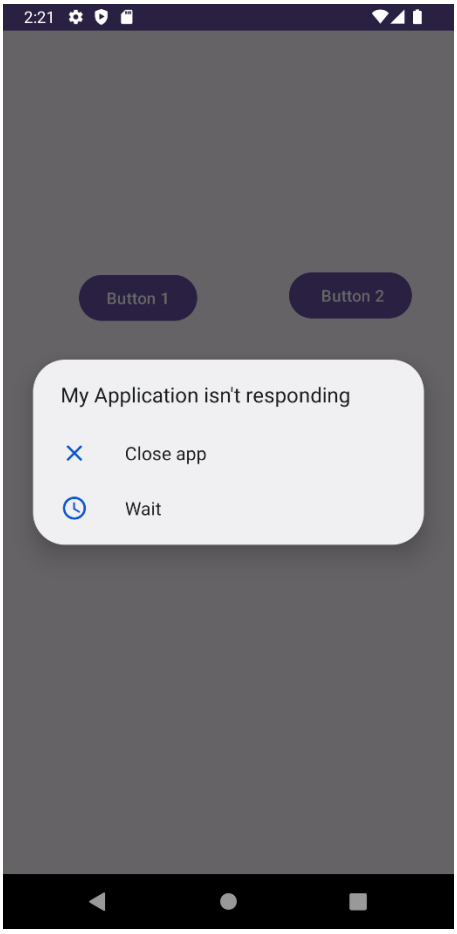
- **getName()** - получить имя потока
- **getPriority()** - получить приоритет потока
- **isAlive()** - определить, выполняется ли поток
- **join()** - ожидать завершение потока
- **run()** - запуск потока. Тут пишется основной код
- **sleep()** - приостановить поток на заданное время
- **start()** - запустить поток

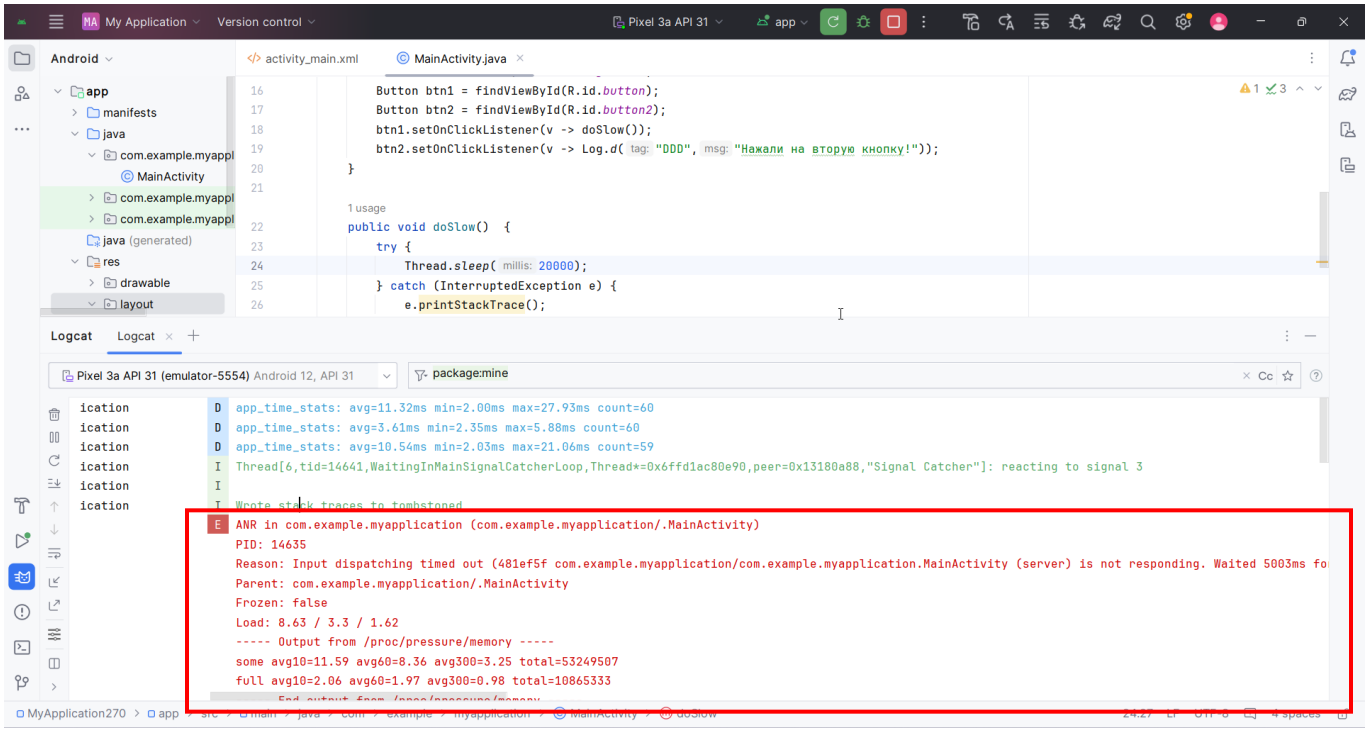
Давайте попробуем реализовать простейшее приложение под Android со следующим кодом активности:

```
 public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button btn1 = findViewById(R.id.button);
        Button btn2 = findViewById(R.id.button2);
        btn1.setOnClickListener(v -> doSlow());
        btn2.setOnClickListener(v -> Log.d("DDD","Нажали на вторую
кнопку!"));
    }

    public void doSlow()  {
        try {
     //Приостанавливаем поток
            Thread.sleep(20000);
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

При нажатии на первую кнопку, интерфейс перестанет отвечать и спустя 5 секунд мы получим ANR ошибку (Application Not Respond).





Условия возникновения ANR ошибки:

- Входные события (кнопки и сенсорные события) не обрабатываются в течение 5 секунд;
- **BroadcastReceiver (onRecieve())** не был обработан в течение указанного времени (foreground — 10 с, background — 60 с);
- **ContentProvider** не завершен в течение 10 секунд.

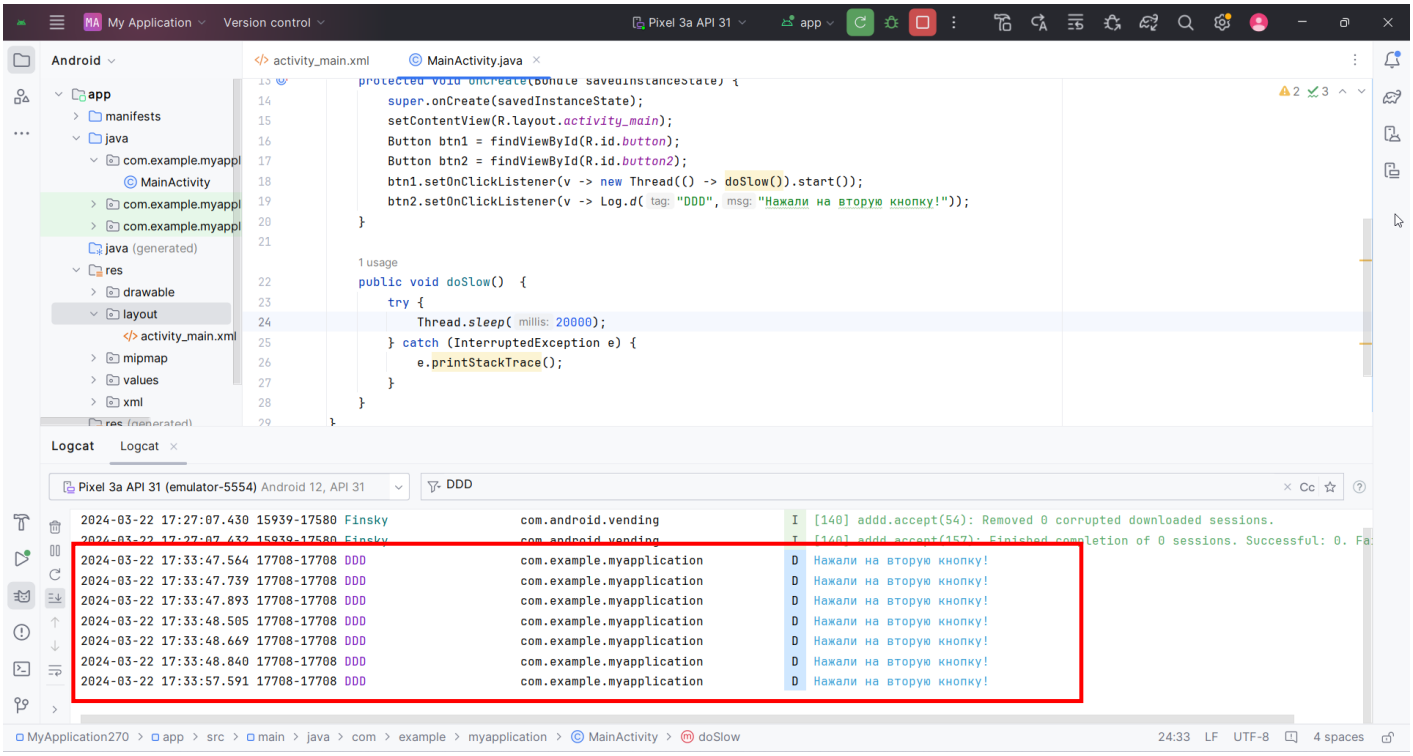
Как избежать ANR ошибки:

- Главный поток пользовательского интерфейса (**UI Thread**) может выполнять логику, связанную только с пользовательским интерфейсом;
- Сложные вычисления (например, операции с базой данных, операции ввода-вывода, сетевые операции и т.д.) производятся в отдельном потоке (**Thread**);
- Используйте **Handler** (подробнее рассмотрим его ниже) для взаимодействия между потоком пользовательского интерфейса и рабочим потоком;
- Используйте **RxJava** и т.д. для обработки асинхронных операций.

Самой простым способом решения такой проблемы в текущем проекте, будет обертка в классический **Thread** вызова метода **doSlow(),** таким образом код активности будет такой:

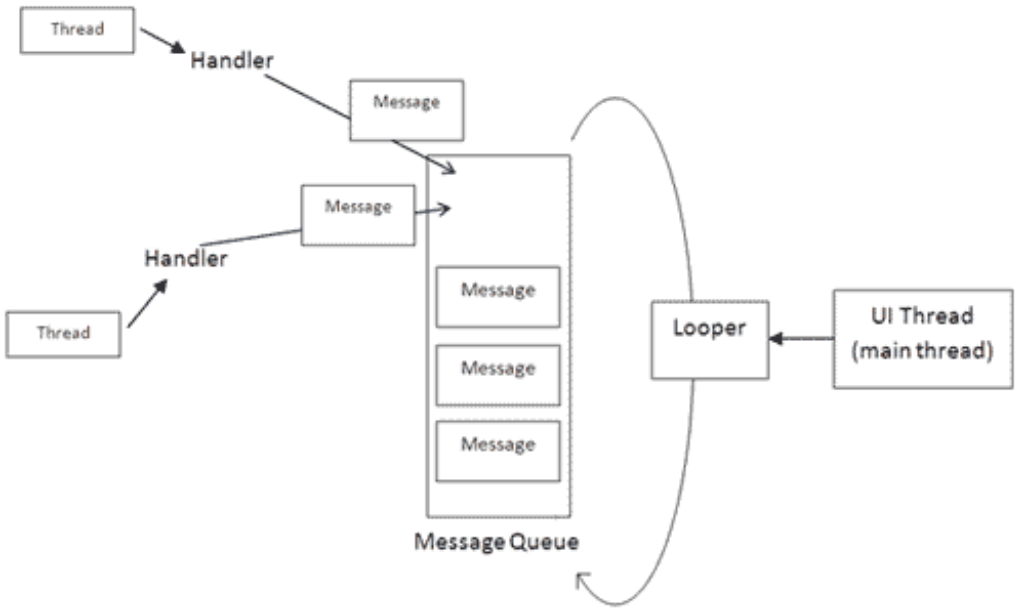
```
 public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button btn1 = findViewById(R.id.button);

        Button btn2 = findViewById(R.id.button2);
             btn1.setOnClickListener(new View.OnClickListener() {
                 @Override
                 public void onClick(View view) {
                     // Определяем объект Thread - новый поток
                     Thread thread = new Thread();
                     doSlow();
                     // Запускаем поток
                     thread.start();
                 }
             });

       // btn1.setOnClickListener(v -> new Thread(() ->
doSlow()).start());
        btn2.setOnClickListener(v -> Log.d("DDD","Нажали на вторую
кнопку!"));
    }

    public void doSlow()  {
        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

При повторном запуске и нажатии на первую кнопку, **ANR** ошибку мы не получим, а кнопка 2 станет доступной и будет откликаться, см. Logcat:



Часть 2. Класс Handler

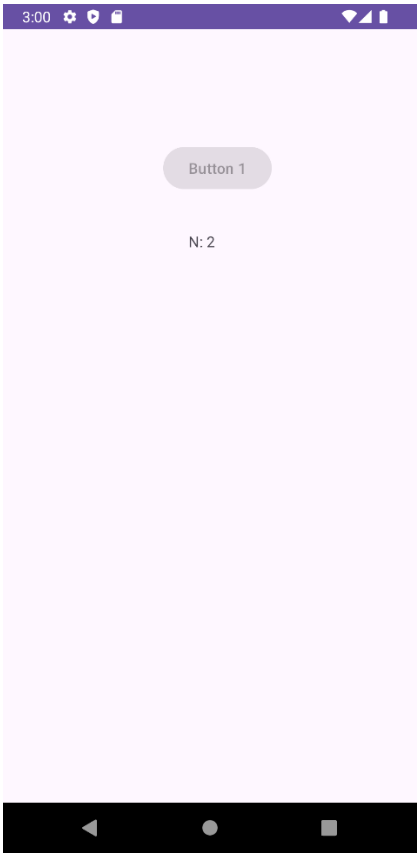
Класс **android.os.Handler** является дальнейшим развитием потоков, упрощающий код. **Handler** может использоваться для планирования выполнения кода в некоторый момент в будущем. Также класс может использоваться для передачи кода, который должен выполняться в другом программном потоке.



Рассмотрим простой пример. Сделаем с вами счетчик, который будет передавать данные в основной поток при помощи **Handler.** Код класса активности:

```
public class MainActivity extends AppCompatActivity {

    private Button btn;
    private Handler handler;
    private TextView tv;

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

        tv = findViewById(R.id.textView);
        // Looper - запускает цикл обработки сообщений
        // и стартует его в главном потоке  - это вызов статического
метода getMainLooper()
        handler = new Handler(Looper.getMainLooper()) {
            @Override
            public void handleMessage(@NonNull Message msg) {
                // здесь мы будем ждать сообщения из другого потока
                int n = msg.getData().getInt("key");
                tv.setText("N: "+n);
                if(n==49) btn.setEnabled(true);
            }
        };

        Button btn = findViewById(R.id.button);
        btn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                btn.setEnabled(false);
                //Создаем поток с помощью интерфейса Runnable
                new Thread(new Runnable() {
```

```
                    @Override
                    //Код для нового потока. Этот поток завершится, когда
метод вернёт управление.
                    public void run() {
                        doSlow();
                    }
                }).start();
            }
        });
    }

    public void doSlow() {
        for(int i=0;i<50;i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            Message message = new Message();
            Bundle bundle = new Bundle();
            bundle.putInt("key",i);
            message.setData(bundle);
            handler.sendMessage(message);
        }
    }
}
```



Часть 3. Класс WorkManager

Одним из современных инструментов организации параллельного выполнения потоков в Android является класс **WorkManager.** С его помощью можно запускать последовательные и параллельные фоновые задачи, получать из них результат, передавать в них данные, отслеживать статус их выполнения и т.п. При этом возможностью запуска параллельных задач инструментарий класса не ограничивается. **WorkManager** – довольно простая, при этом достаточно гибкая библиотека, обладающая множеством дополнительных преимуществ, к которым можно отнести:

- поддержка разовых и периодических асинхронных задач;
- выполнение фоновых задач с учетом ограничений (например, состояние сети, наличие памяти для хранения данных, статус зарядки и др.);
- возможность объединения сложных рабочих запросов в единую цепочку;
- возможность использования выходных данных одной задачи в качестве входных данных для следующей;
- поддержка LiveData для удобного отображения состояния рабочего запроса в пользовательском интерфейсе.

При этом инструментарий класса достаточно простой в использовании.

Давайте рассмотрим пример, в котором используется задержка времени в качестве имитации, к примеру, загрузки файла из интернета. При этом есть возможность запустить процесс скачивания как в потоке, так и без. При нажатии кнопки «Начать не в потоке» процесс загрузки затормозит основной поток, в связи с чем экран устройства станет полностью некликабельным и выведется сообщение ANR. В случае нажатия кнопки «Начать в потоке» процесс загрузки начнется в асинхронном потоке и основной интерфейс не будет заморожен, в том числе CheckBox будет свободно нажиматься.

Создадим проект и в раздел dependencies сборщика gradle добавим следующую строчку:

```
    implementation("androidx.work:work-runtime:2.9.0")
```

После добавления зависимости необходимо синхронизировать изменения, нажав Sync Now.

Актуальною версию можно найти по ссылке: https://developer.android.com/develop/background-work/background-tasks/persistent/getting-started

Приложение будет состоять из одной активности со следующей разметкой:

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

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/btJustDoIt"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Начать в потоке" />

    <Button
        android:id="@+id/btStart"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Начать не в потоке" />

    <CheckBox
        android:id="@+id/checkBox"
        android:layout_width="184dp"
        android:layout_height="wrap_content"
        android:text="проверка" />
</LinearLayout>
```

Для организации параллельного потока необходимо создать класс, который будет наследоваться от класса **Worker** и определить метод **doWork():**

```
public class MyWorker extends Worker {
    public final String TAG = "MY_TAG";

    public MyWorker(@NonNull Context context, @NonNull WorkerParameters
workerParams) {
        super(context, workerParams);
    }
```

```
    @NonNull
    @Override
    public Result doWork() {
        Log.v(TAG, "Work is in progress");
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Log.v(TAG, "Work finished");
        return Worker.Result.success();
    }
}
```

В метод **doWork()** необходимо поместить код, который будет выполнен в параллельном потоке. В данном случае просто используется задержка времени в качестве имитации длительной задачи. Метод возвращает результат **Worker.Result.success(),** означающий успешное выполнение задачи. При этом код метода **doWork()** будет выполнен не в **UI потоке**. Могут быть следующие варианты:

- **Result.success():** Работа успешно завершена.
- **Result.failure():** Работа завершилась неудачей.
- **Result.retry():** Работа завершилась неудачей, и ее следует попробовать в другой раз.

Для создания параллельной задачи в классе активности объект **MyWorker** нужно обернуть в **WorkRequest**, который позволяет нам задать условия запуска и входные параметры к задаче:

```
OneTimeWorkRequest work = new
OneTimeWorkRequest.Builder(MyWorker.class).build();
```

Пока что мы ничего не задаем, а просто создаем **OneTimeWorkRequest**, которому говорим, что запускать надо будет задачу MyWorker.

Есть два типа WorkRequests:

- OneTimeWorkRequest – подразумевает разовое выполнение WorkRequest.

- PeriodicWorkRequest – WorkRequest, который будет повторяться в цикле.

Теперь можно запускать задачу:

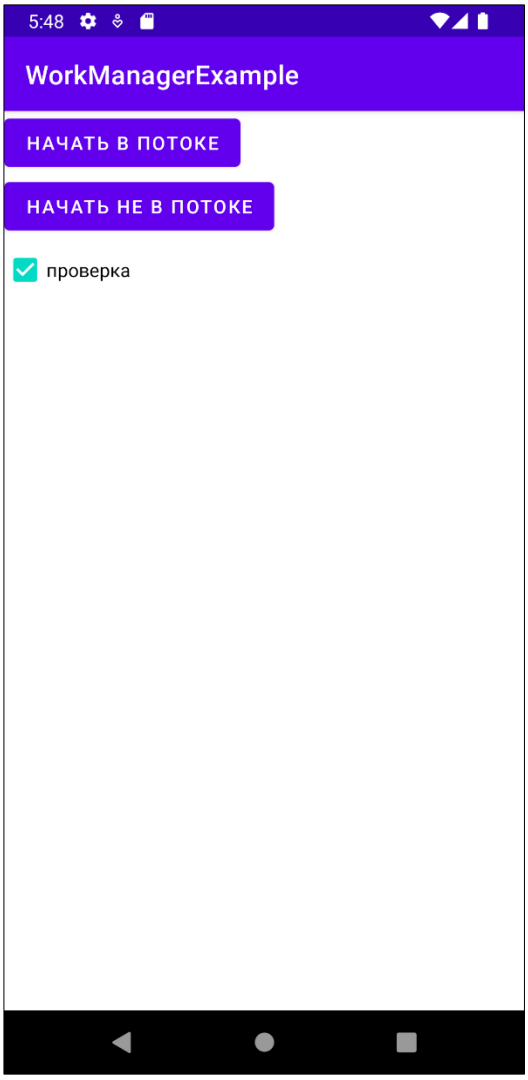
```
WorkManager.getInstance(getApplicationContext()).enqueue(work);
```

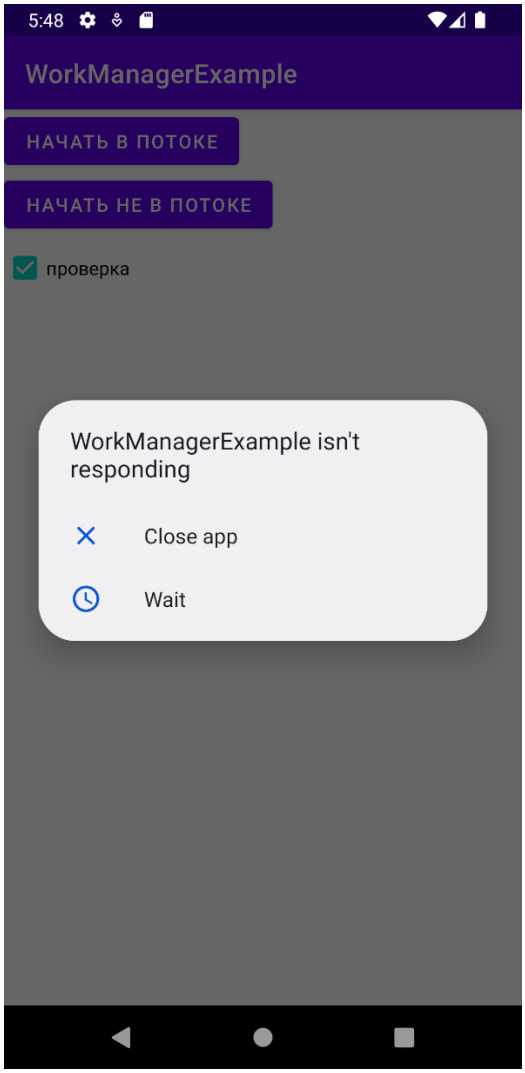
Полностью класс MainActivity выглядит так:

```
public class MainActivity extends AppCompatActivity {
    public final String TAG = "RRR";
    Button bStart, btJustDoIt;

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

    bStart = findViewById(R.id.btStart);
    btJustDoIt= findViewById(R.id.btJustDoIt);

    // устанавливаем обработчик на кнопку "Начать в потоке"
    btJustDoIt.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
        OneTimeWorkRequest work =
          new OneTimeWorkRequest.Builder(MyWorker.class).build();
          WorkManager.getInstance(getApplicationContext()).enqueue(work);
            }
        });

        // устанавливаем обработчик на кнопку "Начать не в потоке"
        bStart.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Log.d(TAG, "Work is in progress");
                try {
                    Thread.sleep(10000);
                } catch (InterruptedException e) {
```

```
                    e.printStackTrace();
                }
                Log.d(TAG, "Work finished");
            }
        });
    }
}
```

Запускаем:





Когда мы запускаем задачу, нам может понадобиться передать в нее данные и получить обратно результат. Для этих целей служит объект типа **Data**. Передать входные данные в задачу можно, например, следующим образом:

```
    Data data = new Data.Builder()
            .putString("key1","Test message")
```

```
          .putInt("key2",123123123).build();
workRequest = new OneTimeWorkRequest.Builder(MyWork.class)
        .setInputData(data)
        .build();
```

Когда задача будет запущена, то внутри ее (в **MyWorker.java**) мы можем получить эти входные данные:

```
     public class MyWork extends Worker {
    public MyWork(@NonNull Context context, @NonNull WorkerParameters
workerParams) {
        super(context, workerParams);
        int x = workerParams.getInputData().getInt("key2",0);
        String s = getInputData().getString("key1");
    }
     …
}
```

Чтобы задача вернула данные, необходимо передать их в метод **success**. Код в **MyWorker.java** будет следующим:

```
     Data data = new Data.Builder()
        .putString("key3","Hello from Worker!")
             .putInt("key0",123)
        .build();
return Result.success(data);
```

Эти выходные данные мы сможем получить из объекта **WorkStatus** в классе **MainActivity**:

```
     WorkManager.getInstance(this).getWorkInfoByIdLiveData(workRequest.ge
tId()).observe(
        this, new Observer<WorkInfo>() {
            @Override
            public void onChanged(WorkInfo workInfo) {
                Log.d("RRR","State = " + workInfo.getState());
                Log.d("RRR",
"key="+workInfo.getOutputData().getString("key3"));
```

```
                    Log.d("RRR",
"key="+workInfo.getOutputData().getInt("key0",0));
            }
        }
);
```

Также, существует возможность запускать параллельно и последовательно набор **WorkRequest-ов**, например:

```
    OneTimeWorkRequest workRequest, workRequest2;
workRequest = new OneTimeWorkRequest.Builder(MyWork.class)
        .setInputData(data)
        .build();
workRequest2 = new OneTimeWorkRequest.Builder(MyWork.class)
        .build();
List<OneTimeWorkRequest> list = new ArrayList<>();
// параллельно
WorkManager.getInstance(this).enqueue(list);
// последовательно 1
WorkManager.getInstance(this).beginWith(list).enqueue();
    // последовательно 2
WorkManager.getInstance(this).beginWith(workRequest).then(workRequest2).e
nqueue();
```

Задание

1) Реализовать запуск последовательно 3 задач.

2) Реализовать запуск 2 задач параллельно.

3) Разработать приложение, которое при нажатии кнопки будет загружать изображение по Rest API: https://random.dog/woof.json и выводить его на экран.

Источники

1) https://developer.android.com/develop/background-work/background-tasks/persistent

2) https://habr.com/ru/articles/439086/

3) https://developer.alexanderklimov.ru/android/theory/handler.php

4) https://startandroid.ru/ru/courses/architecture-components/27-course/architecture-components/562-urok-29-workmanager-vvedenie.html