План занятия

1. Задача
2. Activity
3. Intent'ы
4. Sharing
5. Receiving
6. Explicit
7. Lifecycle
8. Итоги

ЗАДАЧА

ЗАДАЧА

Мы научились визуально стилизовать наши элементы, пришла пора немного поговорить о возможности интеграции с другими приложениями (нас интересует шаринг в другие приложения, например, Telegram, и постинг в наше приложение из других приложений).

ACTIVITY

ACTIVITY

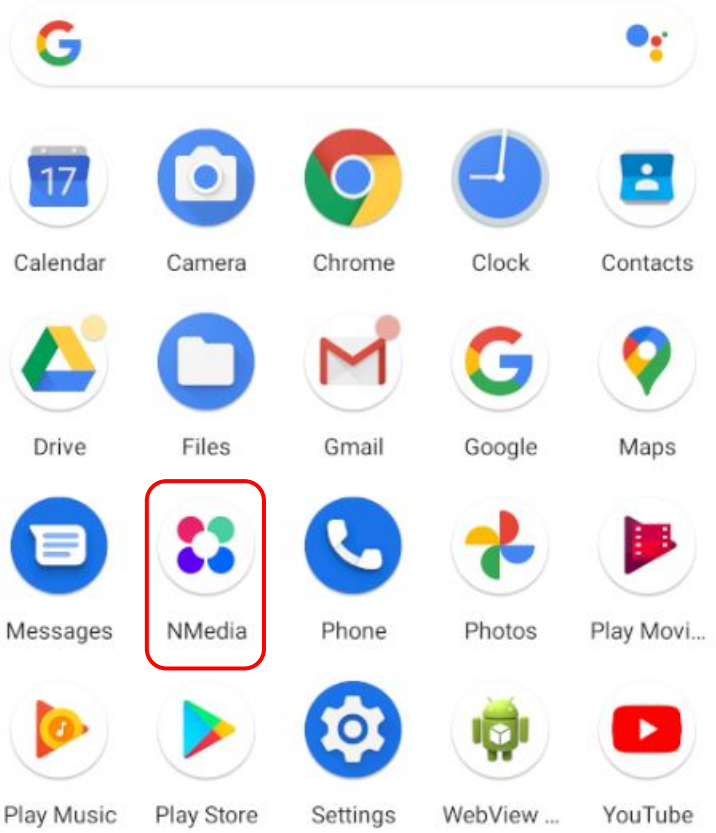
Activity — это один из 4х компонентов Android приложения. В большинстве* случаев это точка входа в UI нашего приложения.

До этого у нас была всего одна Activity, но приложение может состоять из нескольких Activity, между которыми можно перемещаться.

Примечание*: виджеты рабочего стола и notification'ы также предоставляют UI, но Activity не являются.

ACTIVITY

Рассмотрим сначала процесс запуска нашей Activity (пока она у нас одна): мы заходим в Launcher и нажимаем на иконку приложения:



MANIFEST

Activity должны быть описаны в Manifest'е нашего приложения с помощью тега activity:

```
<activity android:name=".activity.MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Сам тег обладает большим количество атрибутов, которые можно настраивать, но ключевой, конечно же, name.

MANIFEST

name должны быть FQN именем (Fully Qualified Name), либо, если имя пакета совпадает с тем, что указан в теге manifest, может начинаться с точки:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="ru.netology.nmedia">


    <activity android:name=".activity.MainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
```

INTENT'ы

INTENT'ы

Компоненты Android (такие как Activity) не запускаются сами по себе. Чтобы активировать какой-то компонент (запустить его), используется такая абстракция как Intent.

Intent

Kotlin | Java

```
open class Intent : Parcelable, Cloneable
```

kotlin.Any
↳ android.content.Intent

INTENT'ы

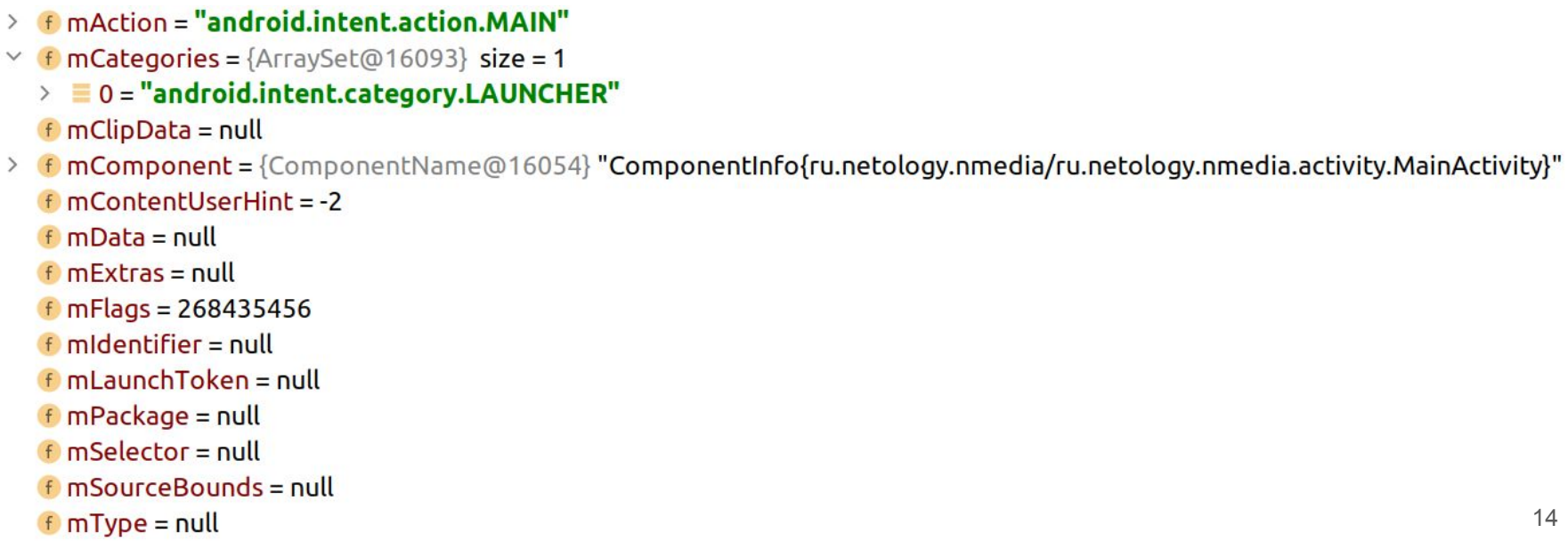
У Intent'а есть несколько ключевых полей (некоторые из них могут быть не заданы):

- action — что надо сделать;
- category — дополнительная информация о типе компонента, который должен обрабатывать Intent;
- component — FQN запускаемого компонента.

INTENT'ы

Фактически, объект Intent'а передаётся от одного компонента Android другому.

Получить Intent можно с помощью property intent (благодаря get/setIntent) или посмотреть в field mIntent:

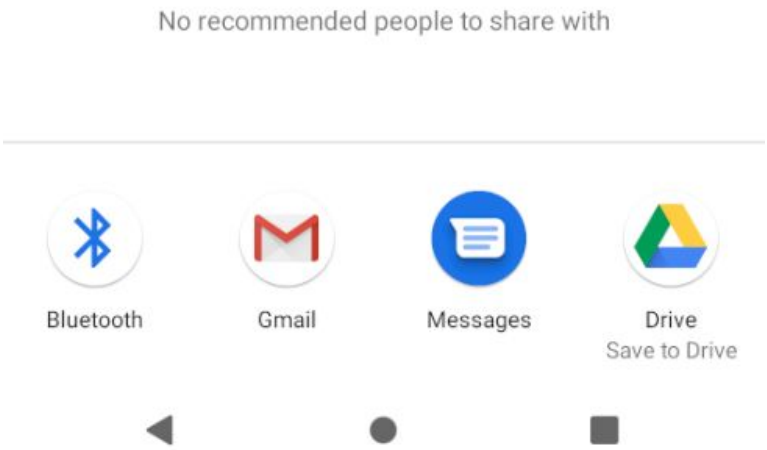


IMPLICIT & EXPLICIT INTENTS

Intent'ы можно разделить на две большие группы:

- Explicit — явные
- Implicit — неявные

В explicit intent'ах явно указывается FQN запускаемого компонента. В implicit он явно не указывается, что позволяет Android формировать диалог выбора:



INTENT FILTER

Для реагирования на implicit intent'ы компоненты объявляют так называемые Intent Filter'ы, согласно которым система и решает, какие именно компоненты показывать (например, некоторые умеют обрабатывать текст, а некоторые — фото).

INTENT FILTER

Указанный Intent Filter позволяет показывать (и запускать) нашу MainActivity из Launcher'а:

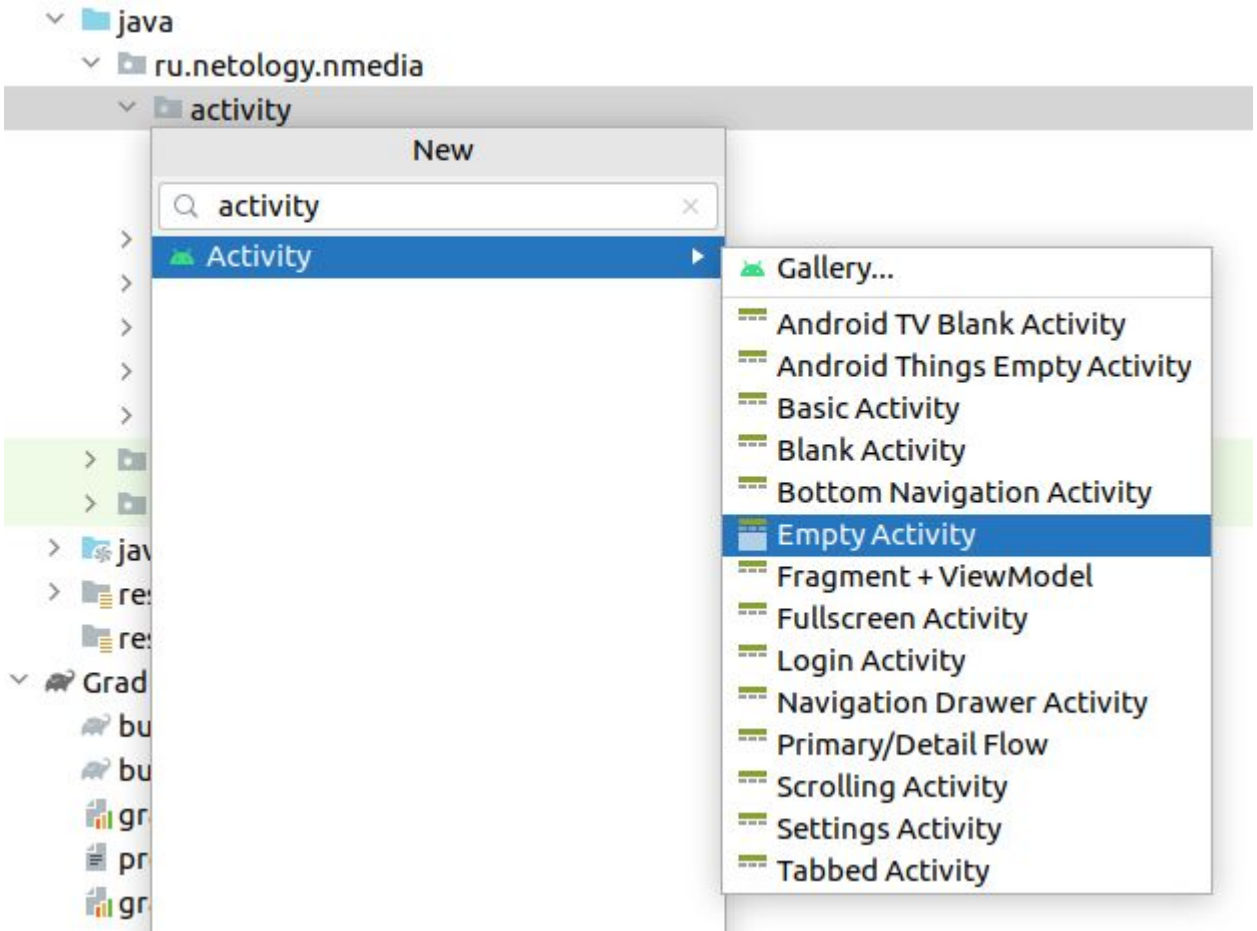
```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />

    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

ACTION.MAIN & CATEGORY.LAUNCHER

```java
/**
 *  Activity Action: Start as a main entry point, does not expect to
 *  receive data.
 *  <p>Input: nothing
 *  <p>Output: nothing
 */
@SdkConstant(SdkConstantType.ACTIVITY_INTENT_ACTION)
public static final String ACTION_MAIN = "android.intent.action.MAIN";

/**
 * Should be displayed in the top-level launcher.
 */
@SdkConstant(SdkConstantType.INTENT_CATEGORY)
public static final String CATEGORY_LAUNCHER = "android.intent.category.LAUNCHER";
```

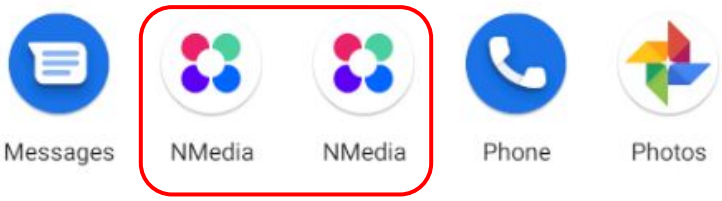
EXPERIMENT

Если мы проведём эксперимент и создадим вторую Activity (удобно её создавать с помощью спец.меню, сразу создастся и layout):



EXPERIMENT

И поставим обеим Activity (MAIN + LAUNCHER — так обычно не делают):



То в Launcher'е мы увидим целых две иконки нашего приложения, каждая из которых будет запускать свою Activity.

SHARING

SHARING

Ok, эксперименты — это здорово, но как сделать что-то полезное?

Давайте начнём с шаринга: мы хотим иметь возможность отправлять текст наших постов в другие приложения (любые, которые смогут принимать текст).

В документации на Intent смотрим стандартные Action'ы для Activity и находим там ACTION_SEND.

SHARING

Дальше всё достаточно просто: в документации описано, как реализовать необходимую нам функциональность для разных вариантов:

- текст
- бинарные данные

MIME-TYPE

Одна из ключевых вещей — это указания MIME-типа. Это просто строка, которая описывает тип передаваемых данных, например:

- text/plain
- image/png
- и т.д.

CHOOSER

Поскольку бинарных данных у нас пока нет, будем передавать просто текст нашего поста:

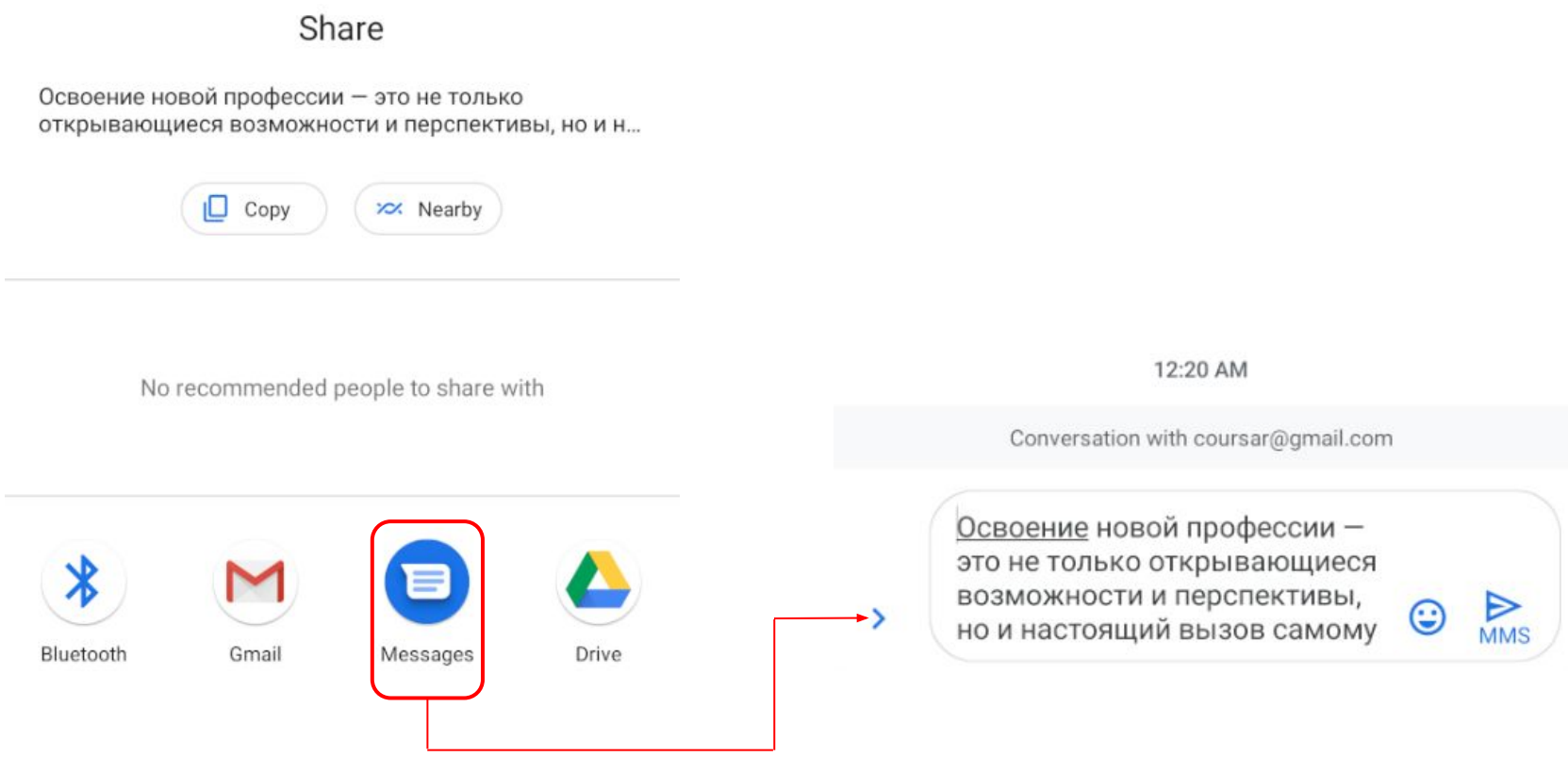
```
override fun onShare(post: Post) {
    val intent = Intent().apply { this: Intent
        action = Intent.ACTION_SEND
        putExtra(Intent.EXTRA_TEXT, post.content)
        type = "text/plain"
    }

    val shareIntent = Intent.createChooser(intent, getString(R.string.chooser_share_post))
    startActivity(shareIntent)
}
```

Логика:

1. Создаётся intent на отправку текста.
2. Создаётся intent на показ Chooser'a (меню выбора).
3. startActivity приводит к запуску компонента Activity через выбор.

CHOOSER

На самом деле, createChooser — это просто хелпер для создания Intent'a с ACTION_CHOOSER. При запуске получим:



PARCELABLE & BUNDLE

Здесь отдельно стоит упомянуть то, как именно передаются данные через putExtra: в Android существует специальный класс Bundle (key — value), который «сериализуется» и десериализуется при передаче.

RECEIVING

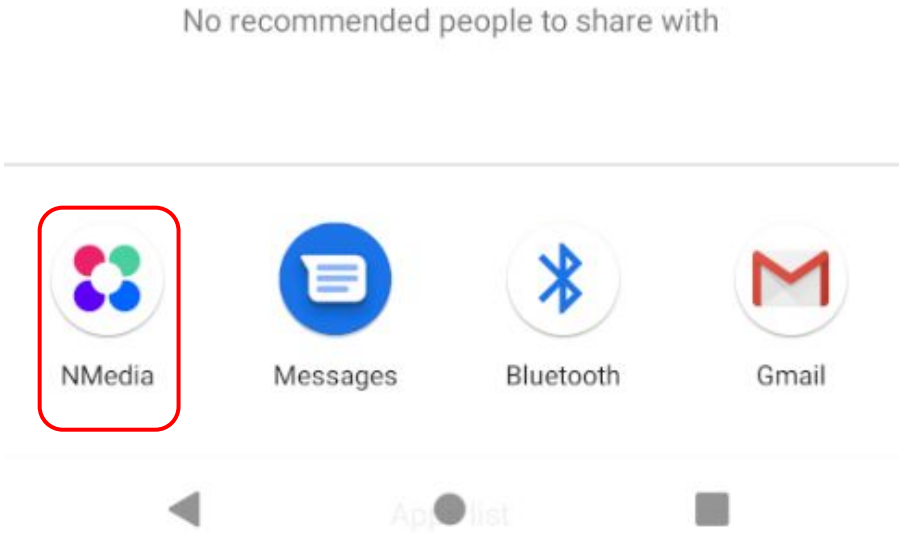
RECEIVING

Отправлять мы научились, а как получать? Аналогично, мы должны объявить Intent Filter на получение данных:

```
<activity android:name=".activity.IntentHandlerActivity">
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" /> ←обязательно для implicit
        <data android:mimeType="text/plain" />
    </intent-filter>
</activity>
```

Обратите внимание, что не обязательно конкретно указывать MIME-тип, можно указать и text/* или даже */*, что будет означать любой MIME-тип (другой вопрос, что вам нужно будет писать обработчики).

RECEIVING


No recommended people to share with
NMedia
Messages
Bluetooth
Gmail

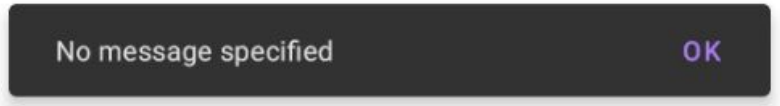
RECEIVING

```
intent?.let { it: Intent
    if (it.action != Intent.ACTION_SEND) {
        return@let
    }

    val text = it.getStringExtra(Intent.EXTRA_TEXT)
    if (text.isNullOrBlank()) {
        Snackbar.make(binding.root, R.string.error_empty_content, LENGTH_INDEFINITE)
            .setAction(android.R.string.ok) { it: View!
                finish()
            }
            .show()
        return@let
    }
    // TODO: handle text
}
```

завершаем работу Activity

SNACKBAR

В прошлом приложении мы использовали Toast. В этом же показываем Snackbar:



И по нажатии на OK — выходим (здесь мы специально продемонстрировали использование системных ресурсов через android.R.string.ok — в большинстве случаев лучше так не делать и хранить всё в собственных ресурсах).

По поводу завершения работы Activity: всё зависит от бизнес-логики приложения, возможно, вы не захотите, чтобы она не завершалась.

RECEIVING

Ключевые моменты:

1. Проверяем тип Intent'a.
2. Всегда проверяем сами данные (в данном случае нам другое приложение может прислать всё, что угодно).
3. Если пришло что-то не то, оповещаем пользователя, чтобы он не думал, что проблема на нашей стороне.

EXPLICIT

EXPLICIT

Мы посмотрели на работу с Implicit Intent'ами. Давайте теперь посмотрим на работу с Explicit.

Для этого немного переделаем архитектуру нашего приложения: сделаем добавление не в той же Activity, а в новой.

ACTIVITY RESULT CONTRACT

Для создания Intent и обработки возвращаемого результата создадим класс NewPostResultContract

```
class NewPostResultContract : ActivityResultContract<Unit, String?>() {

    override fun createIntent(context: Context, input: Unit?): Intent =
        Intent(context, NewPostActivity::class.java)

    override fun parseResult(resultCode: Int, intent: Intent?): String? =
        if (resultCode == Activity.RESULT_OK) {
            intent?.getStringExtra(Intent.EXTRA_TEXT)
        } else {
            null
        }
}
```

ACTIVITY RESULT CONTRACT

Зарегистрируем функцию, которая будет вызвана при завершении NewPostActivity

```
val newPostLauncher = registerForActivityResult(NewPostResultContract()) { result ->
    result ?: return@registerForActivityResult
    viewModel.changeContent(result)
    viewModel.save()
}
```

вызовется после parseResult

ACTIVITY RESULT CONTRACT

Для корректной работы данного механизма следует убедиться, что используются библиотеки activity и fragment* версий не ниже 1.2.1 и 1.3.1

```
def fragment_version = "1.3.1"
def activity_version = "1.2.1"
implementation "androidx.fragment:fragment-ktx:$fragment_version"
implementation "androidx.activity:activity-ktx:$activity_version"
```

Примечание*: о фрагментах мы поговорим подробнее на одной из следующих лекций

FAB

Для UI мы задействуем специальный компонент, который называется FAB (Floating Action Button):

```
binding.fab.setOnClickListener {
    newPostLauncher.launch()
}
```

Здесь мы создаём Explicit Intent, указывая, объект какого класса его должен обрабатывать. И мы не только запускаем Activity, мы ещё запускаем его с требованием вернуть нам назад результат (т.е. тот текст, который введёт пользователь).

ВОЗВРАТ ЗНАЧЕНИЯ

Для возврата значений в Activity также используется механизм Intent'ов:

1. Мы создаём Intent
2. Наполняем его необходимыми данными
3. Кладём в setResult
4. Вызываем finish

При этом в качестве кодов возврата чаще всего используют следующие:

```
/** Standard activity result: operation canceled. */
public static final int RESULT_CANCELED    = 0;
/** Standard activity result: operation succeeded. */
public static final int RESULT_OK           = -1;
```

ВОЗВРАТ ЗНАЧЕНИЯ

```
class NewPostActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val binding = ActivityNewPostBinding.inflate(layoutInflater)
        setContentView(binding.root)
        binding.edit.requestFocus();
        binding.ok.setOnClickListener { it: View!
            val intent = Intent()
            if (binding.edit.text.isBlank()) {
                setResult(Activity.RESULT_CANCELED, intent)
            } else {
                val content = binding.edit.text.toString()
                intent.putExtra(Intent.EXTRA_TEXT, content)
                setResult(Activity.RESULT_OK, intent)
            }
            finish()
        }
    }
}
```

PUTEXTRA

Важно: putExtra определяет имя ключа, по которому нужно положить значение. Не обязательно это должно быть одно из статических полей класса Intent. Вы можете сделать собственное значение (например, с помощью companion object), главное, чтобы оба Activity договорились, какое:

```
public @NonNull Intent putExtra(String name, @Nullable String value) {...}
```

APPBAR

В качестве разметки мы использовали AppBar из MDC вместе с CoordinatorLayout (это такой готовый Layout для ряда взаимодействий).

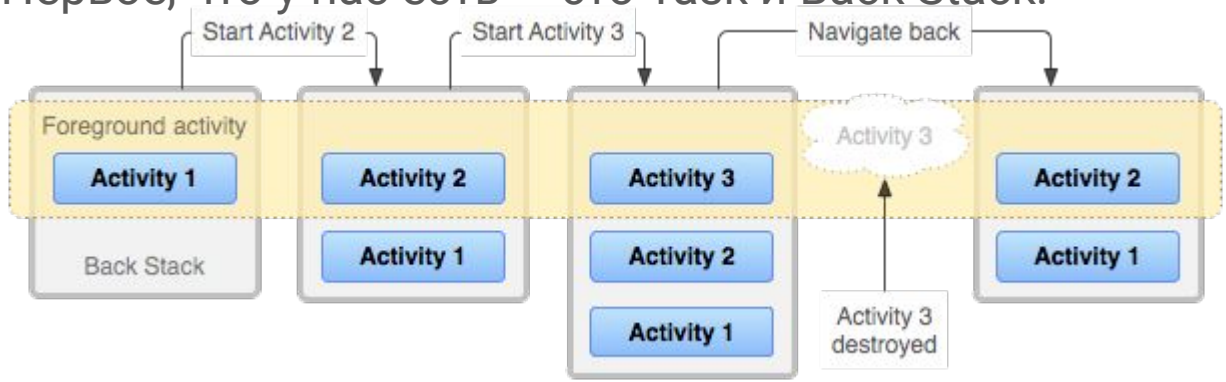
LIFECYCLE

LIFECYCLE

Теперь, когда мы организовали перемещение между Activity с помощью Intent'ов, возникает вопрос: а как оно на самом деле устроено, и что происходит с самой Activity?

TASK & BACK STACK

Первое, что у нас есть — это Task и Back Stack:



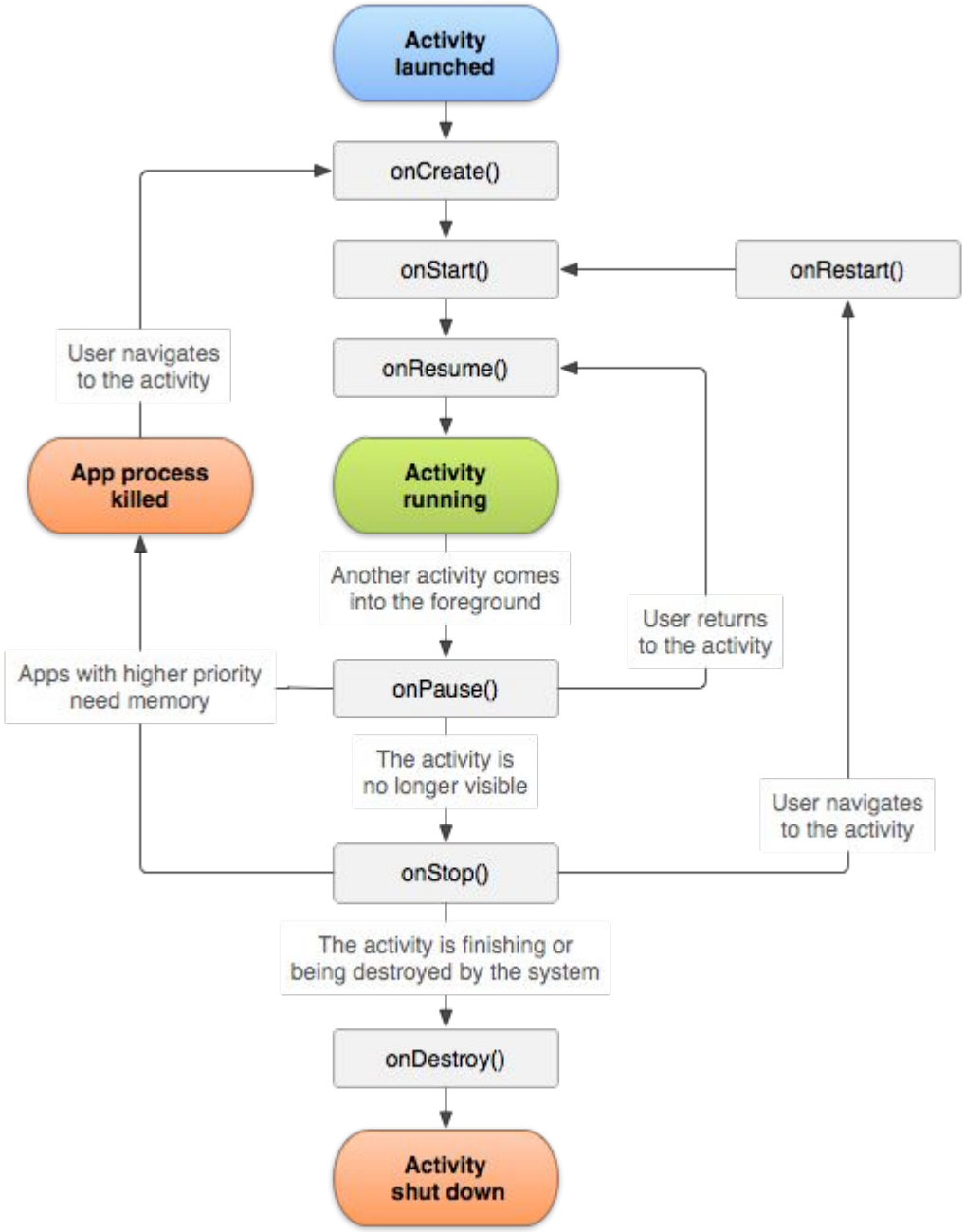
Источник: developer.android.com

Когда из Activity 1 запускается Activity 2, то первая Activity уходит в «фон» (background), а вторая переходит в режим foreground.

Когда пользователь нажмёт на кнопку назад, или будет вызван finish, текущая Activity завершит свою работу и предыдущая снова перейдёт в foreground.

LIFECYCLE

Но что значит «переходит в режим»? На самом деле, Android будет вызывать на Activity определённые member functions и отрисовывать/либо не отрисовывать Activity.



Источник: developer.android.com

ONCREATE

Функция, которая вызывается тогда, когда система создает наше Activity.

В onCreate выполняется базовая логика приложения, которая происходит один раз за весь жизненный цикл, например, привязка xml-разметки или связывание с ViewModel.

Этот метод получает в качестве параметра savedInstanceState, содержащий в себе данные, которые перед этим были сохранены. Если Activity не существовало до этого, то этот параметр пустой (null).

ONSTART & ONRESUME

onStart — Activity становится видимым для пользователя. В этой функции приложение может инициализировать код, который работает с UI.

onResume — Activity может взаимодействовать с пользователем. После вызова этой функции Activity остается в «активном» состоянии до тех пор, пока что-то не произойдет, и Activity не потеряет фокус. Например, при телефонном звонке, или если пользователь перейдет на другую Activity, или будет заблокирован/выключен экран.

ONPAUSE & ONSTOP

onPause — вызывается системой, когда пользователь покидает Activity (однако это не всегда означает, что Activity будет уничтожено).

onStop — Activity больше не видно пользователю. Это может произойти, например, когда новое Activity запущено и занимает весь экран. Также система может вызвать onStop, когда Activity собирается завершиться. Необходимо освободить ресурсы, которые больше не требуются. Например, остановить анимацию.

ONDESTROY

onDestroy — вызывается перед тем, как Activity будет уничтожено.

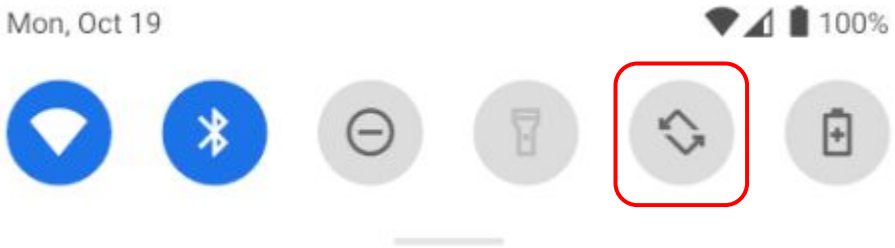
Система вызывает этот метод если:

- Activity завершается,
- система временно уничтожает наше Activity при изменении конфигурации (поворот экрана).

CONFIGURATION CHANGE

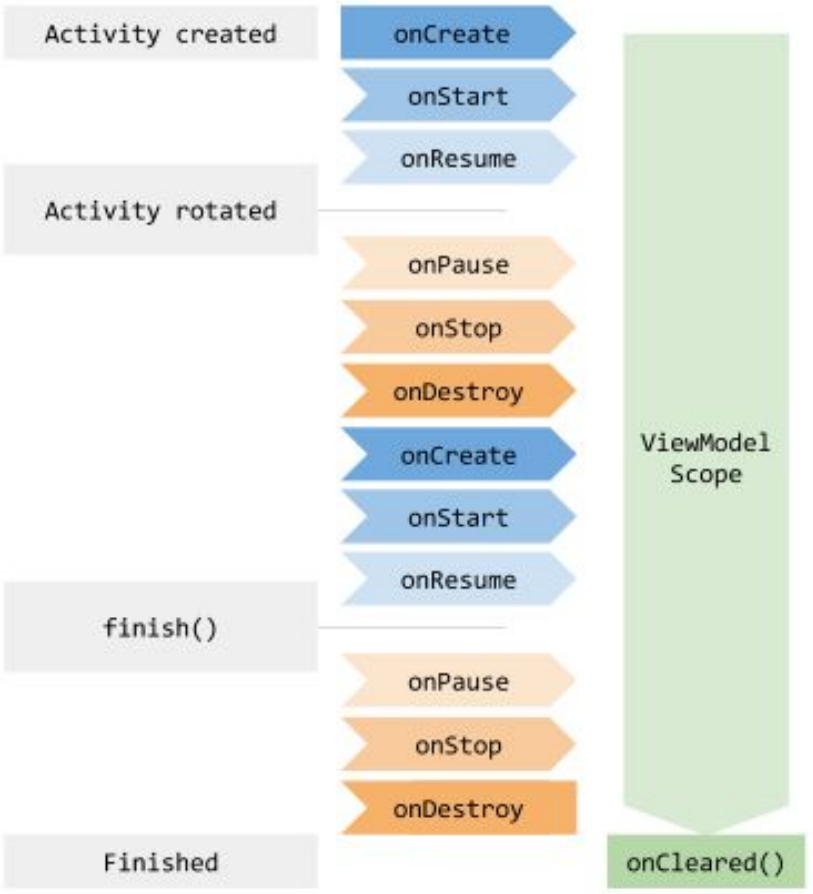
Интересно, это значит, что если мы перевернём экран, то Activity пересоздастся? Давайте залайкаем одну запись и попробуем перевернуть экран.

Не забудьте включить автоповорот экрана в настройках устройства:



CONFIGURATION CHANGE

Данные сохранятся, хотя вызовется и onDestroy, и onCreate. Почему так? Дело в том, что ViewModel имеет другой жизненный цикл, и это позволяет нам не задумываться о том, какие изменения конфигурации происходят (за счёт onCreate Activity снова подпишется на LiveData и заново получит список наших постов).



Источник: developer.android.com

MEMORY RECLAIM

Отдельно стоит отметить, что в Android существует механизм очистки памяти, который позволяет убивать процессы (именно в процессах работают наши Activity), неиспользуемые в данный момент пользователем для высвобождения ресурсов.

Android просто сохраняет скриншот последнего состояния экрана и только когда вы его (приложение) снова активируете, воссоздаёт процесс и запускает Activity. В таких случаях ViewModel нас не спасёт и придётся сохранять состояние либо в постоянном хранилище (БД, SharedPreferences, файлы и т.д.), либо воспользоваться onSaveInstanceState, который позволит сохранить необходимую информацию в Bundle.

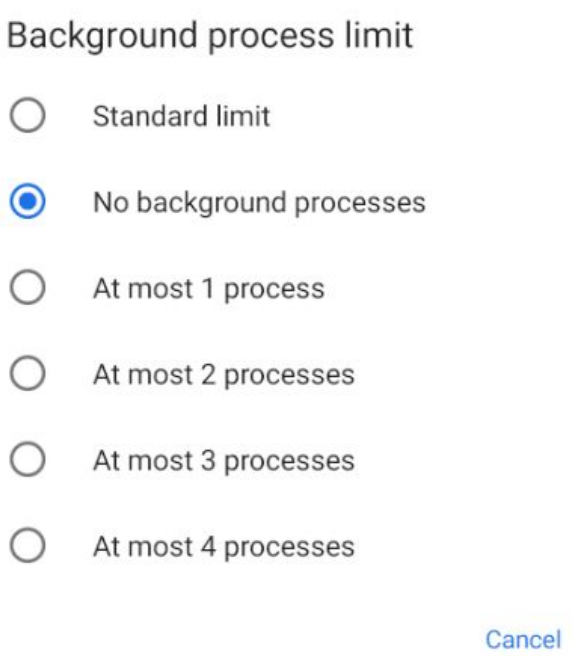
SAVEINSTANCESTATE

Для полей ввода, имеющих ID, и ряда других подобных, не нужно в Bundle сохранять их текущее значение (Android это сделает — сохранит и восстановит самостоятельно).

Полученный Bundle затем будет передан в onRestoreInstanceState и onCreate.

ВАЖНО

Чтобы продемонстрировать «убийство» процесса, достаточно перейти в настройки разработчика* { } Developer options и включить опцию:



Теперь при переходе в другой процесс (например, при шаринге), наш процесс (и Activity в нём) уничтожится.

Примечание*: как включить этот режим, описано в документации.

ПРИОРИТЕТ

Порядок, в соответствии с которым Android решает, какой процесс убить, основан на приоритете и описан в документации.

Ключевое: тестируйте своё приложение в режиме No Background Process, чтобы ваше приложение было готово к этому.

ИТОГИ

ИТОГИ

Сегодня мы обсудили вопросы взаимодействия между компонентами (как внешними, так и внутренними) с помощью Intent'ов и обсудили жизненный цикл Activity.