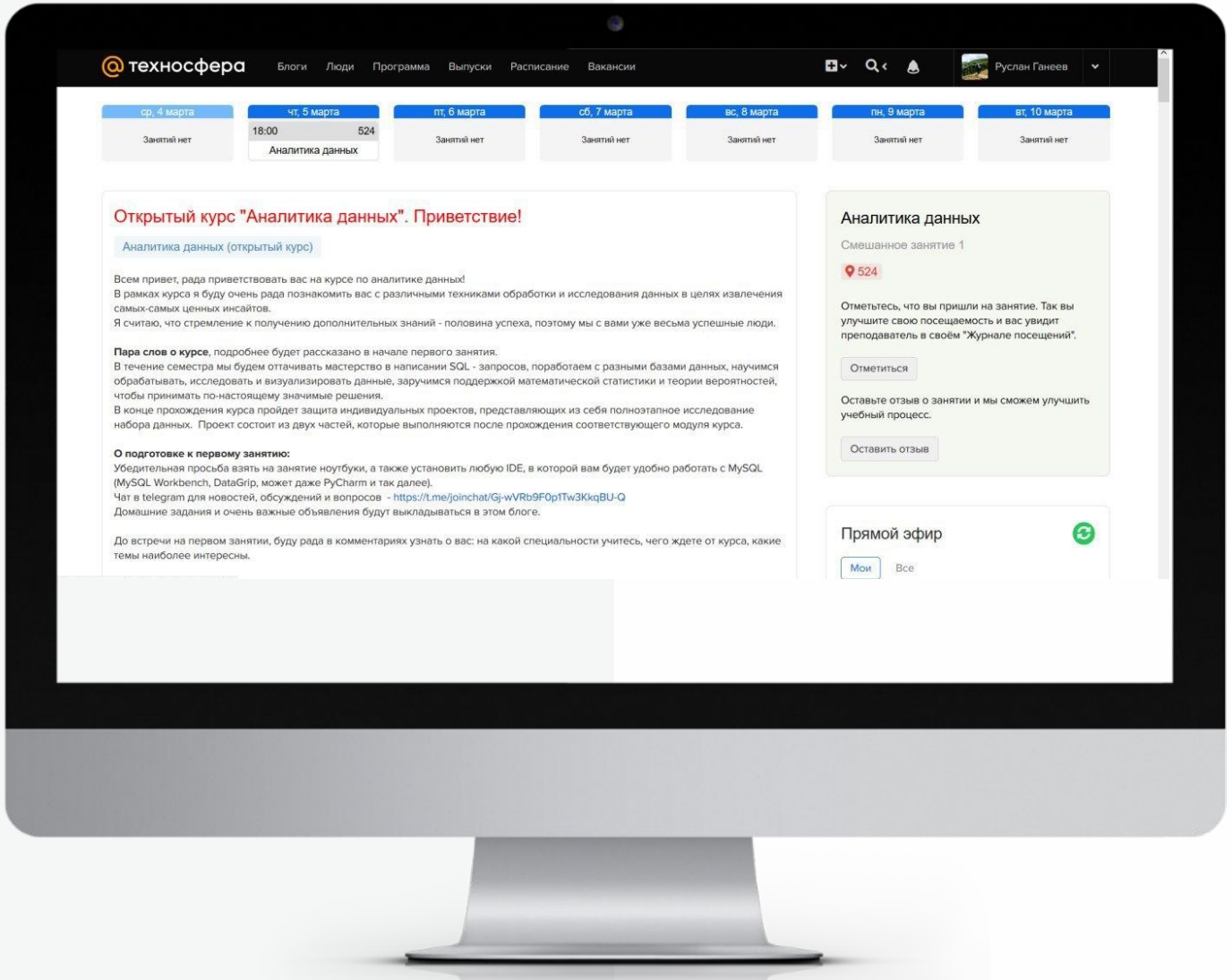
x mail.ru group

Activity. Fragment. Lifecycle

Клещин Никита



Напоминание отметиться на портале

Немного обо мне

Клещин Никита

- В Android разработке с 2010 года
- Работал в компании **Afisha&Rambler (Rambler&Co)**
- “Зарождал” мобильную разработку в компании **lamoda**
- Руководил Android разработкой в компании **Delivery Club**
- Моя команда занимались секретными проектами в **Mail.Ru**
  - Один из этих проектов - **Смотри mail.ru**
- Сейчас работаю над проектом **Premier One**

Кратко

Что помним?

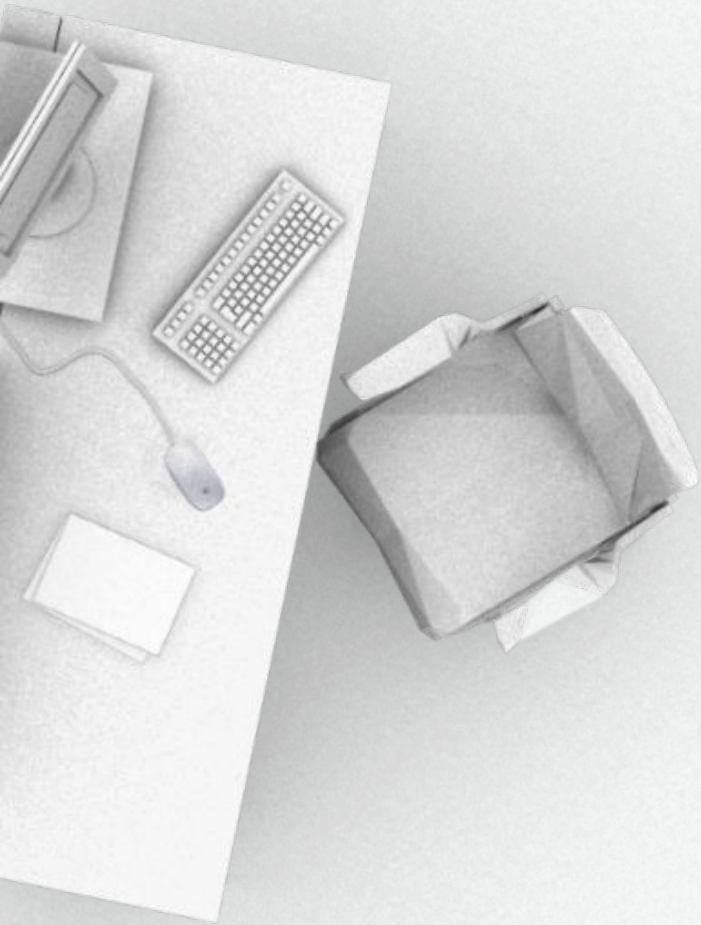
- Activity
- View/ViewGroup

Что успели сделать?

- Как ваша работа с проектом?

Содержание занятия

1. Activity
2. Intent и Activity
3. Fragment
4. Lifecycle

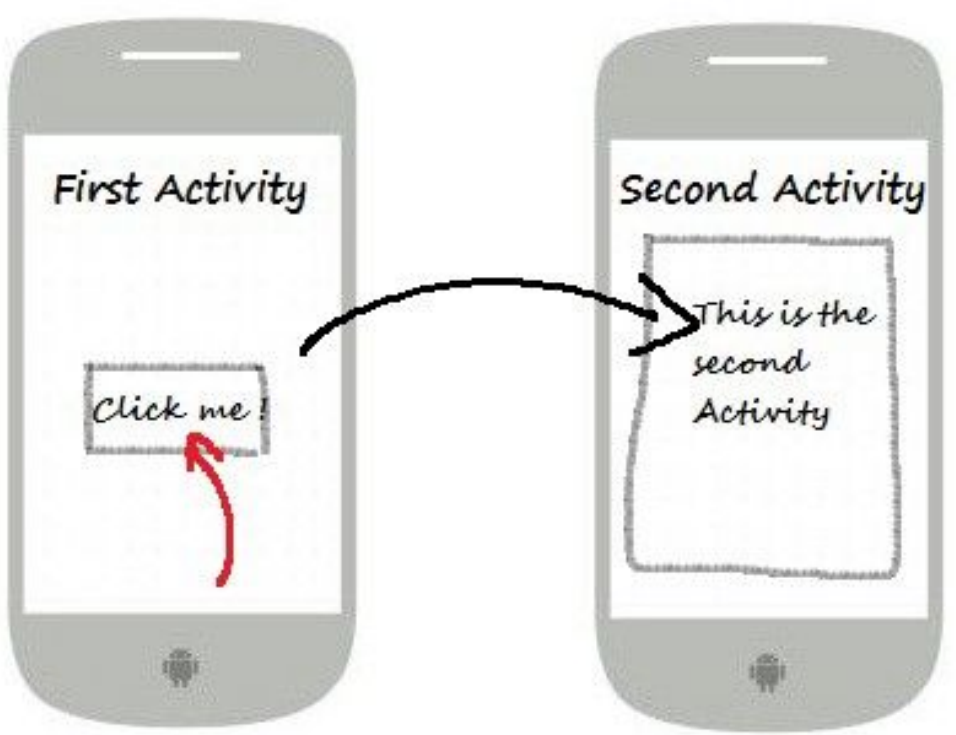


Activity

Один из главных основных компонентов приложения

Что это?

- Входит в список основных компонентов приложения
- Отвечает за визуальную часть приложения
- Отвечает за взаимодействие с пользователем
- Повышает шансы вашего приложения не быть убитым системой:)
- Это **Context** с доступом к **Window**
- **Context** это… глобальная информация к данным приложения и окружения… но это уже другая история



Из чего состоит Activity?

Код

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

или

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val anyView: View = ...
        setContentView(anyView)
    }
}
```

Верстка

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Регистрация в Manifest

<activity> - тэг для описания **Activity** в манифесте:

- **name** - путь до класса **Activity**
- **theme** - собственная тема, если она должна отличаться от основной темы
- и еще много других параметров...

<activity-alias> - линк на **<activity>**. Единственная его функция - это “красиво” разделить саму **Activity** и точку входа в него.

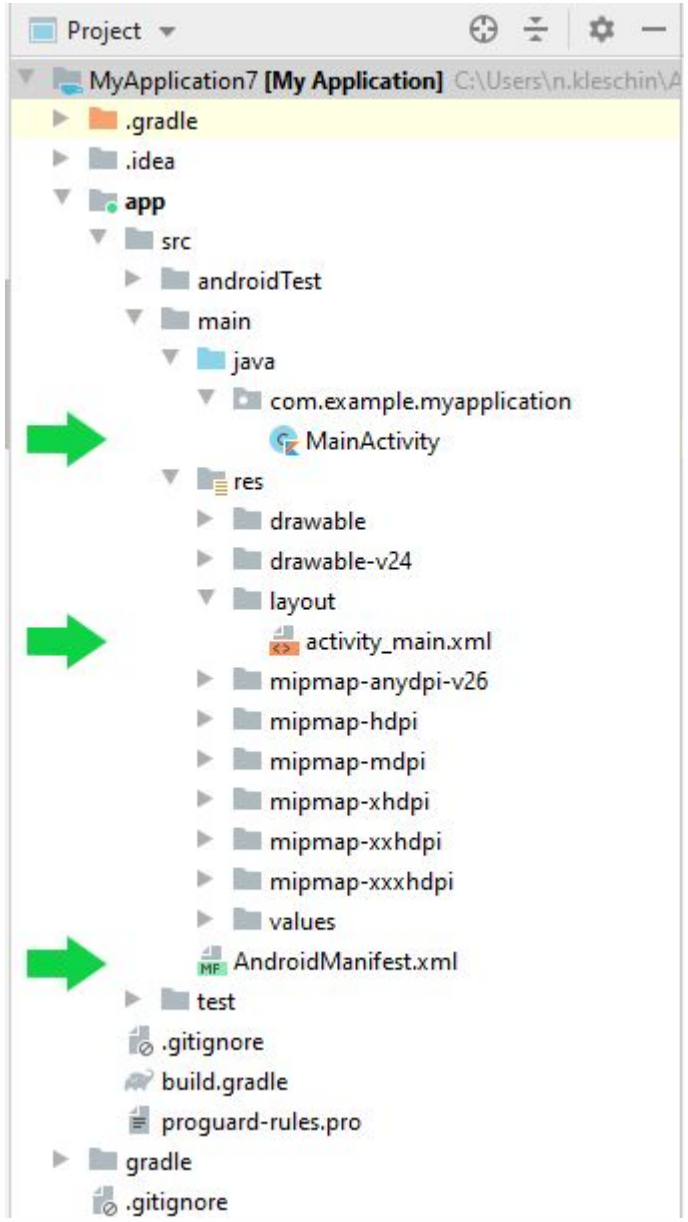
```
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="ru.test.myapplication"
    >

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.MyApplication"
        >

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


        <activity-alias android:name="Deeplink" android:targetActivity=".MainActivity">
            <intent-filter>
                <data android:scheme="myscheme" />
            </intent-filter>
        </activity-alias>
    </application>
</manifest>
```

Надо запомнить



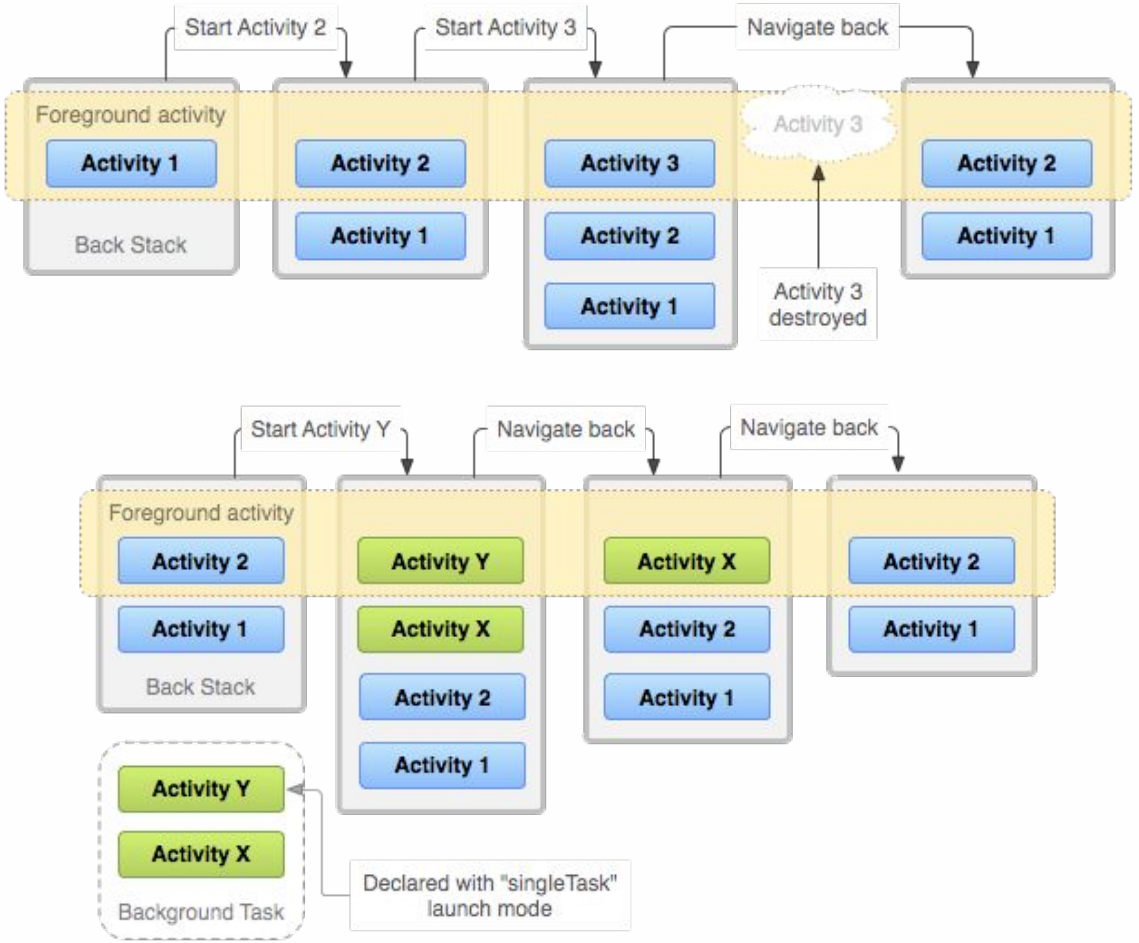
Tasks and Back Stack

Back Stack имеет стратегию **LIFO**

Атрибут `android:launchMode`:

- **standard** - Новый экземпляр при каждом запуске
- **singleTop** - Новый экземпляр, когда **Activity** не на вершине стека
- **singleTask** - Если **Activity** не создана, то будет создана в новом **Task**-е. Иначе - она поднимется из старого **Task**, а все что выше нее, в **Stack**-е, очистится.
- **singleInstance** - Существует всегда в отдельном **Task**

*Если **Activity** переиспользуется, то у нее вызывается метод **onNewIntent()**, вместо **onCreate()**



Изменение конфигурации

Система обрабатывает сама изменений конфигураций (переворот экрана, изменение локали и т.п.) - просто убивает вашу **Activity**:)

Если мы не желаем смерти **Activity**, то можем в манифесте обозначить, какие изменения конфигураций будем обрабатывать самостоятельно.

При помощи параметра `android:configChanges`. Тогда, вместо убийства Activity, система вызовет метод **Activity.onConfigurationChanged.**

```
<activity
    android:name=".MainActivity"
    android:supportsPictureInPicture="true"
    android:configChanges="orientation
        |locale|layoutDirection
        |screenSize|smallestScreenSize|screenLayout"
    />
```

Picture In Picture

Частный случай мультиоконного UI - в данном случае, это небольшой экран поверх экрана;

На телефонах доступен с 26 API (Android 8.0)

Режим **PiP** можно вызвать целенаправленно, либо перейти в него при сворачивании или закрытии **Activity**

Как настроить PiP?

manifest

```
<activity
    android:name=".MainActivity"
    android:supportsPictureInPicture="true"
    android:configChanges="orientation
        |locale|layoutDirection
        |screenSize|smallestScreenSize|screenLayout"
    />
```

В коде **Activity**

```
override fun onPictureInPictureModeChanged(
    isInPictureInPictureMode: Boolean
    , newConfig: Configuration?
    ) {
    ...
}

override fun onUserLeaveHint() {
    if (hasPipFeature()) {
        val params = PictureInPictureParams.Builder().build()
        enterPictureInPictureMode(params)
    }
}


protected fun hasPipFeature(): Boolean {
    return Build.VERSION.SDK_INT >= Build.VERSION_CODES.O
            &&
packageManager.hasSystemFeature(PackageManager.FEATURE_PICTURE_IN_PICTURE)
}

protected fun showPipManually() {
    val params = PictureInPictureParams.Builder().build()
    enterPictureInPictureMode(params)
}
```

Триггеры для запуска Activity

Для запуска активити внутри приложения, гугл рекомендует использовать явное намерение вида `Intent(context, SecondActivity::class.java)`

Однако, для запуска **Activity** извне - потребуется использовать “неявный вызов” при помощи `<intent-filter>`:

- Внутри фильтра надо будет указать действие `<action>`.
- Можно добавить реагирование на какой-нить линк `<data>`.
- Указать категория (говорится что не обязательно, но на практике...) `<category>`.

Intent и Activity

В переводе на русский… “Намерение”

Что это?

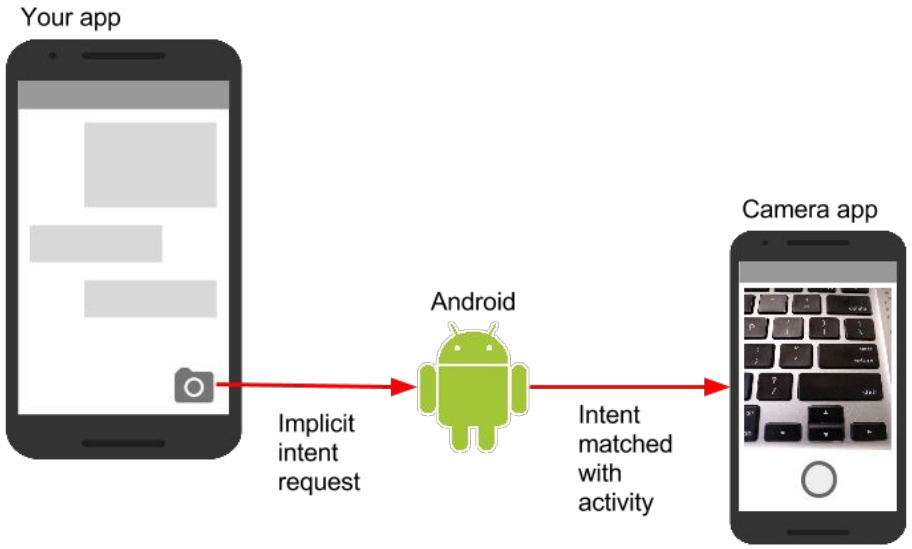
Объект для описания операции для исполнения его системой.

Explicit intent - Явное “намерение”. Указываем класс, к которому хотим обратиться.

- `Intent(context, SecondActivity.class)`

Implicit intent - Неявное “намерение”. Указываем данные, а далее система собирает список обработчиков

- `Intent(Intent.ACTION_VIEW, Uri.parse(url))`
- ```
  Intent(Intent.ACTION_CALL)
      .setData(Uri.parse("tel:555-555-5555"))
  ```


  ```

Передача данных?

Для передачи не стандартизированных параметров, используется объект **Bundle.** У **Intent** есть методы, которые обращаются к **Bundle**

- **intent.putExtra** - одно название, разные параметры

Есть ограничения:

- Можно передавать только примитивы, строки, **Parcelable** и **Serializable** объекты
- Есть ограничение на вес **Bundle** (нет алгоритма, чтобы понять какой именно)

Для чего используется

Запуск **Activity** (про это и поговорим):

- **startActivity** - запустить **Activity**
- **startActivityForResult** - запустить **Activity** с ожиданием результата

Запуск **Service** (службы...):

- **startService** - отправить команду в **Service**

Доставка сообщений, при помощи **BroadcastReceiver**

- **sendBroadcast** - отправить сообщение
- **sendOrderedBroadcast** - отправить сообщение... по порядку

Примеры использования

Отправить письмо

```
val intent = Intent(Intent.ACTION_SENDTO).apply {
   data = Uri.parse("mailto:")
   putExtra(Intent.EXTRA_EMAIL, addresses)
   putExtra(Intent.EXTRA_SUBJECT, "subject")
}
```

Выбрать контакт

```
val intent = Intent(Intent.ACTION_PICK).apply {
   type = ContactsContract.Contacts.CONTENT_TYPE
}
```

Выбрать файл

```
val intent = Intent(Intent.ACTION_OPEN_DOCUMENT).apply {
   addCategory(Intent.CATEGORY_OPENABLE)
}
```

Показать на карте

```
val intent = Intent(Intent.ACTION_VIEW).apply {
   data = "geo:0,0?q=55.8036198,37.409378(LiveHere)".toUri()
}
```

Вызвать звонилку

```
val intent = Intent(Intent.ACTION_DIAL).apply {
   data = Uri.parse("tel:+78001234567")
}
```

Wi-Fi настройки

```
val intent = Intent(Settings.ACTION_WIFI_SETTINGS)
```

Обработка результата

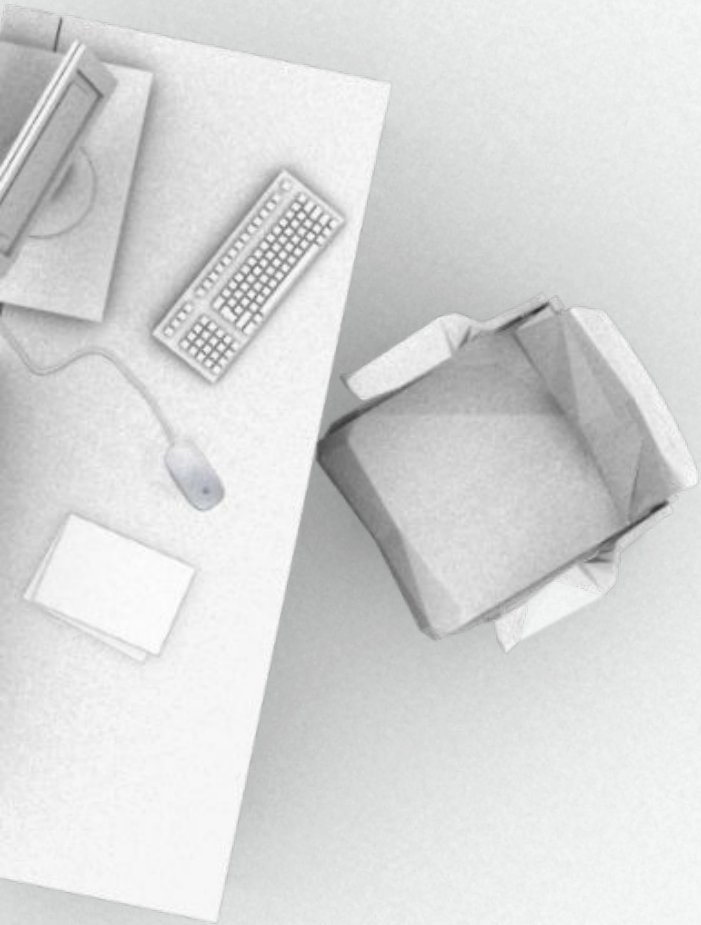
Если нам надо просто запустить следующий экран, то можно используется метод **startActivity()**, если же мы хотим запустить экран, и обработать результат исполнения, то используем метод **startActivityForResult()**.

И переопределить метод **onActivityResult()**, что бы обработать результат исполнения.

```
.setOnClickListener {
    val intent = Intent(Intent.ACTION_OPEN_DOCUMENT).apply {
        addCategory(Intent.CATEGORY_OPENABLE)
    }

    startActivityForResult(intent, PICK_FILE_CODE)
}
```

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    when (requestCode) {
        PICK_FILE_CODE -> handleResult(resultCode, data)
        else -> super.onActivityResult(requestCode, resultCode, data)
    }
}
```



Fragment

Fragment-ируем:)

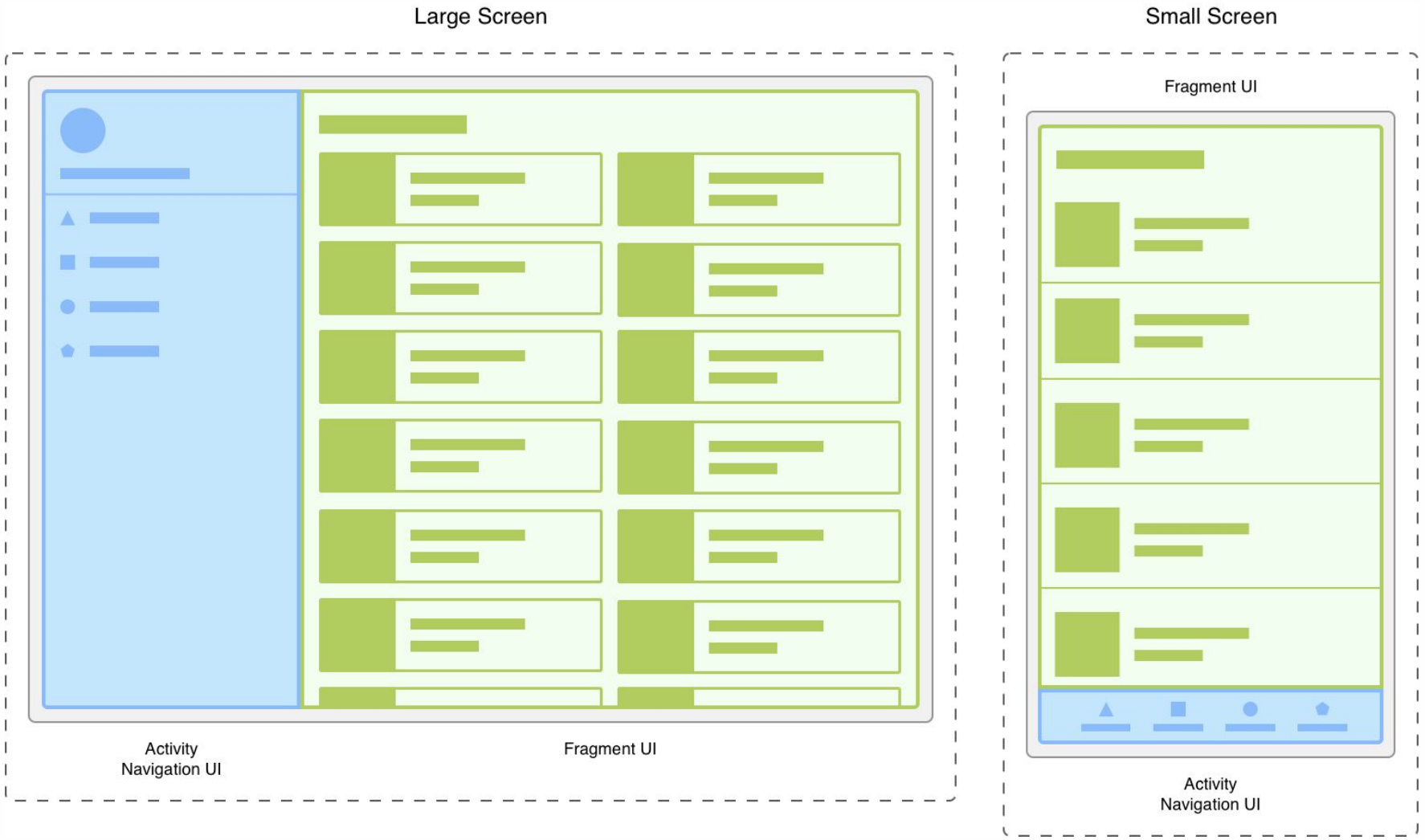
Что это?

Модульные, переиспользуемые (?) части пользовательского интерфейса. Они не самостоятельны - зависят от **Activity**.

Основные плюсы:

- Экран можно разбить на части, и одну из этих частей, со всей логикой, можно переиспользовать на других экранах
- Можно собирать один большой экран из отдельных мелких экранов (то что на телефоне может быть разными экранами, на планшете можно сделать одним экраном)
- Чистит ресурсы при попадании в стэк (в отличии от **Activity**)

Картинка для привлечения внимания



Из чего состоит?

```
class MyFragment: Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? {
        return inflater.inflate(R.layout.content_main, container, false)
    }
}
```

или

```
class MyFragment: Fragment() {
    val contentView: View = ...

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? {
        return contentView
    }
}
```

... Верстка делается так же, как и в **Activity**

Как использовать?

Статическая инициализация

В верстке

```
<fragment
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:name="ru.example.myapplication.MyFragment"
   android:layout_height="match_parent"
   android:layout_width="match_parent"
   />
```

Подразумевается что этот фрагмент не будет заменяться

Динамическая инициализация

В коде

```
supportFragmentManager
   .beginTransaction()
   .replace(R.id.container, MyFragment())
   .commit()
```

В данном случае, создаем транзакцию, для того, чтобы подменить фрагмент, который находится во **View** с идентификатором *container*

Транзакция

FragmentManager - главный компонент для управления фрагментами.
FragmentTransaction - транзакция, для внесения изменения стэка фрагментов

```
.beginTransaction() // Создать транзакцию

    .add() // добавить
    .remove() // удалить
    .replace() // заменить

    .setTransition() // анимация переходов (из имеющихся)
    .setCustomAnimations() // анимация перехода (своя)
    .setSharedElement() // для анимации “перемещения” View

    .addToBackStack() // добавить запись в стэк

    .commitAllowingStateLoss() // commit(), закончить транзакцию
```

Стэк

- **FragmentManager** принадлежит **Activity**
- **FragmentManager** контролит стэк фрагментов

- Можно посмотреть элементы стэка:
  - **getFragments()**
  - **findFragmentByTag()**
  - **findFragmentById()**

- Работа с записями:
  - **addOnBackStackChangedListener()** - подписаться на изменения стэка
  - **getBackStackEntryCount()** - количество записей
  - **getBackStackEntryAt(index)** - взять запись по индексу
  - **popBackStack()** - убрать верхний элемент

Сами фрагменты

Основные

Fragment - самый обычный вариант. Все что описано применимо к нему

DialogFragment - для отображения диалогов (через метод **show**). Но так же умеет все то, что и **Fragment**

Специализированные (не видел чтобы использовали их)

ListFragment - заточен под **ListView**

PreferenceFragment - заточен под **<PreferenceScreen>**

Еще был такой **WebViewFragment** - работал с **WebView**

*Для работы с **FragmentManager** нужен **FragmentActivity**

Best Practice! Создания фрагмента в коде

Информацию для инициализации в фрагмент можно передать при помощи метода **setArguments(Bundle)**. Конструктор у фрагмента лучше не перегружать

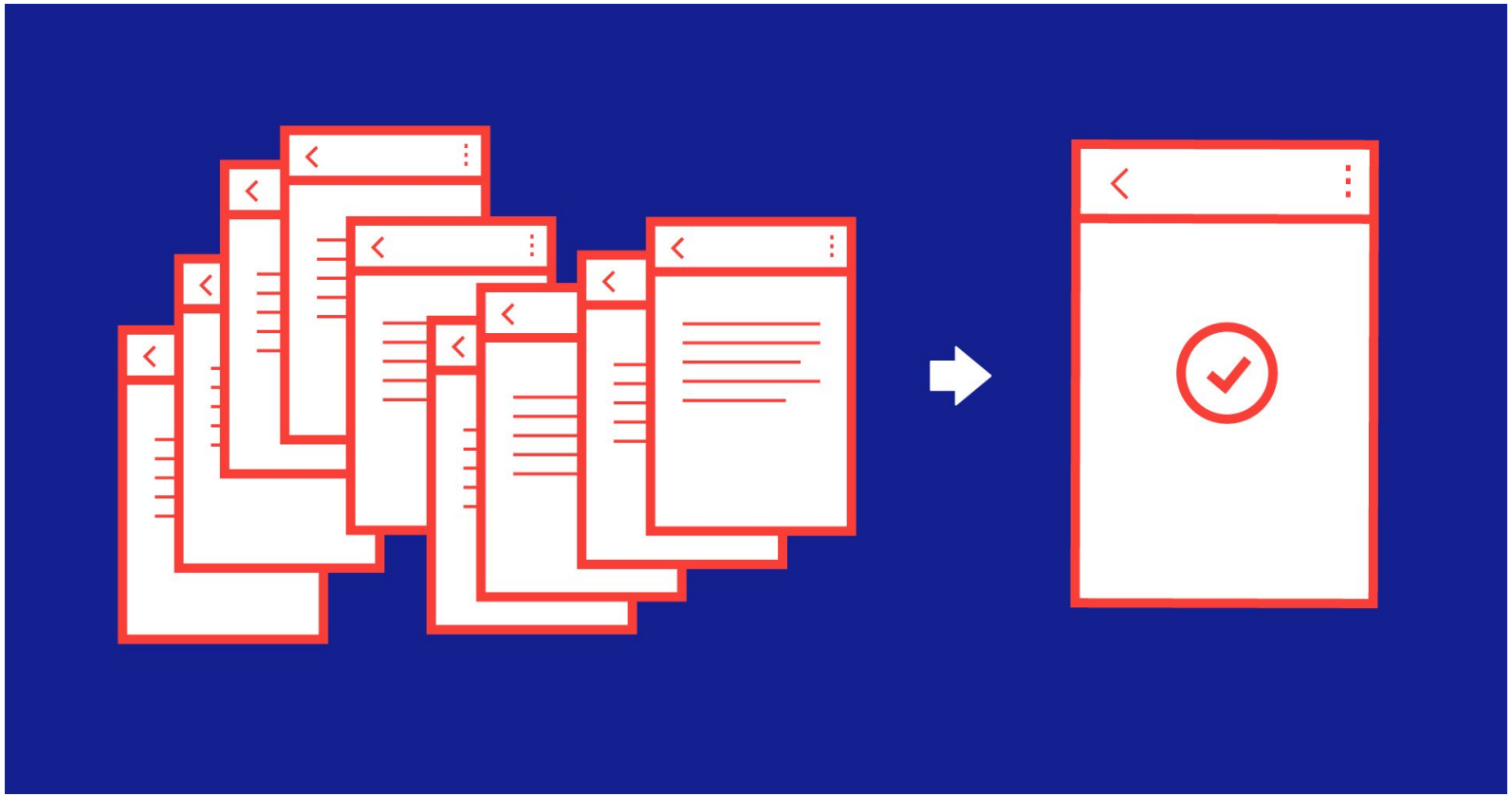
```
fun newInstance(droid: Droid): DroidDetailsFragment {
    val extras = Bundle().apply {
        putSerializable(EXTRAS_DROID, droid)
    }

    val fragment = DroidDetailsFragment().apply {
        arguments = extras
    }

    return fragment
}
```

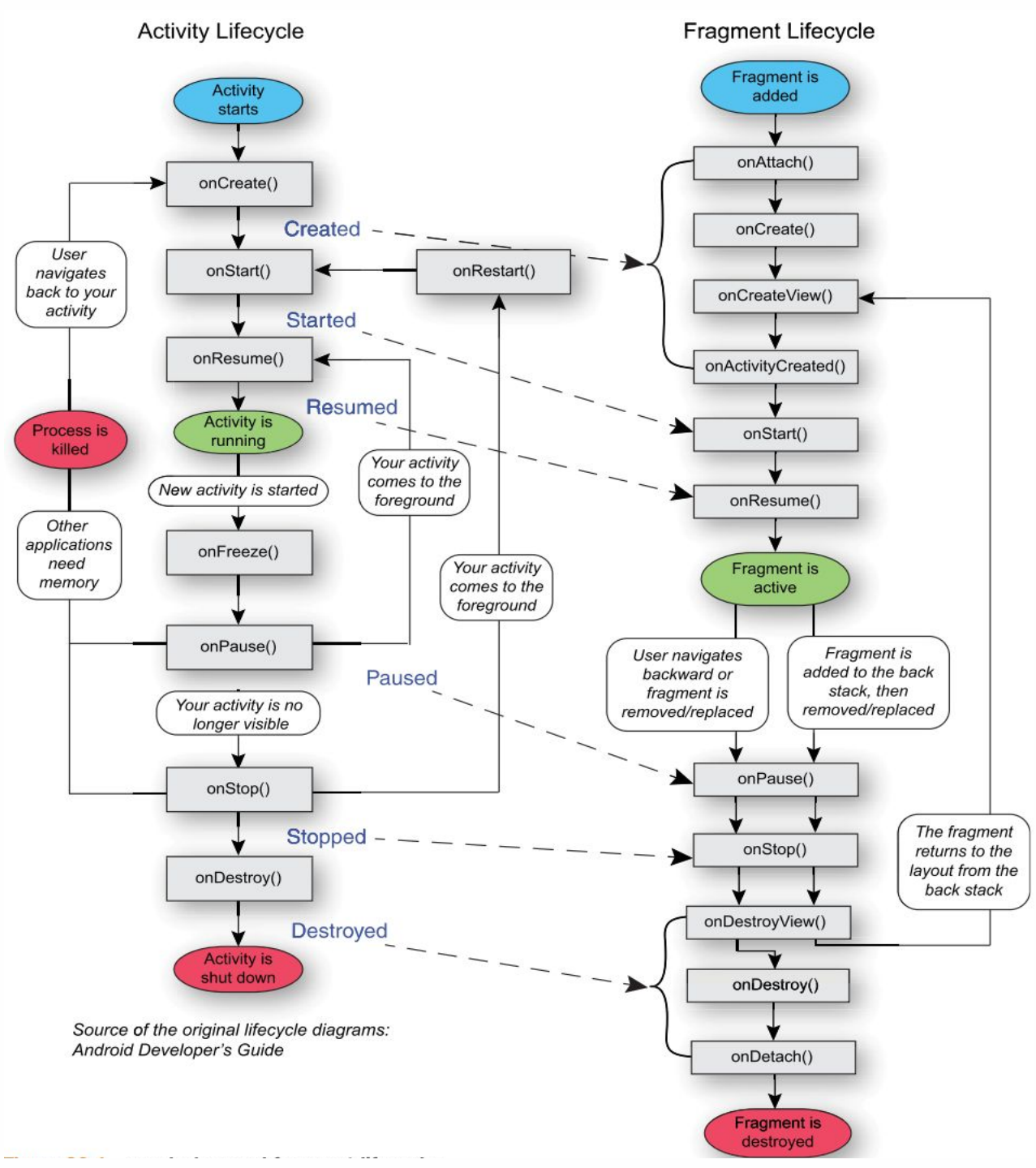
Hello SingleActivity

Поскольку фрагменты тоже можно класть в стэк, почему бы тогда не попробовать писать приложение с одной **Activity** с кучей фрагментов?



Lifecycle

Жизненный цикл



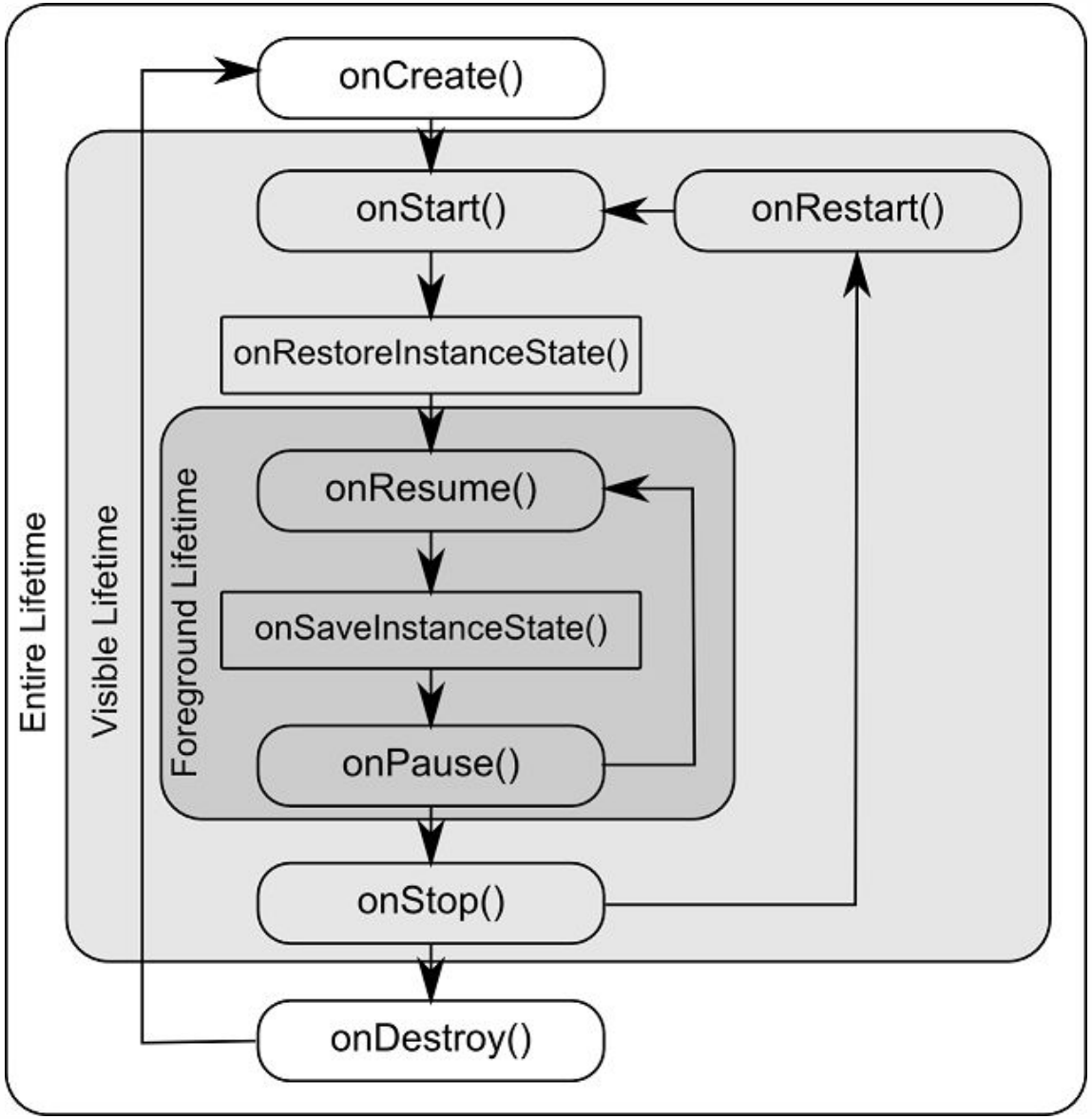
Source of the original lifecycle diagrams: Android Developer's Guide

onCreate - **onDestroy** - начало и конец жизненного пути компонента:)

onStart - onStop - период видимости компонента

onResume - onPause - период активности компонента

onSaveInstanceState - намек системой, сохранить какие-нибудь данные для восстановления состояния



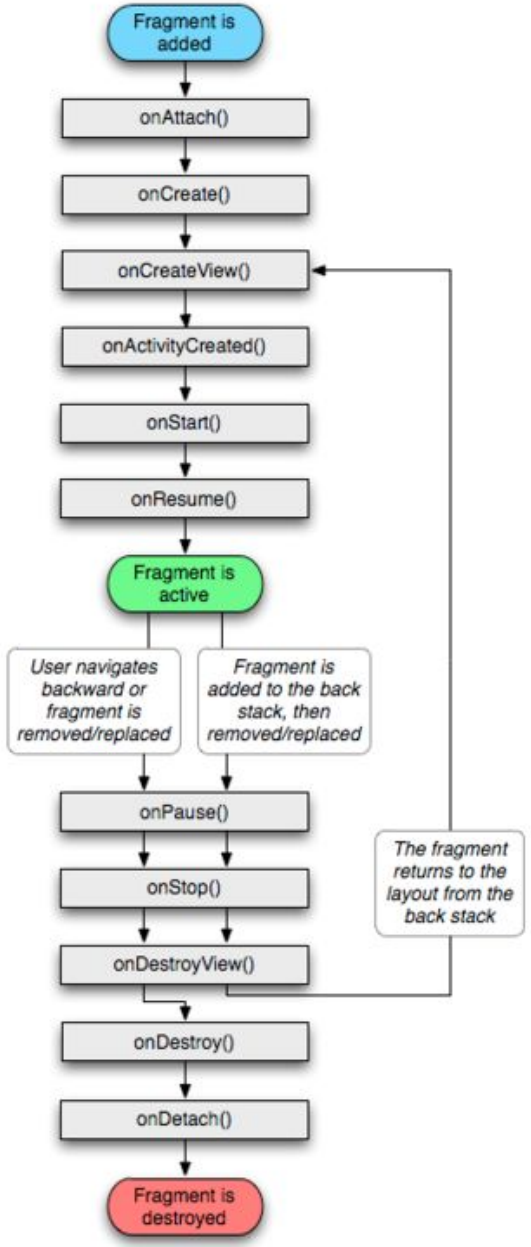
У фрагмента жизненный цикл немного длиннее, но его методы все равно соотносятся с методами **Activity**.

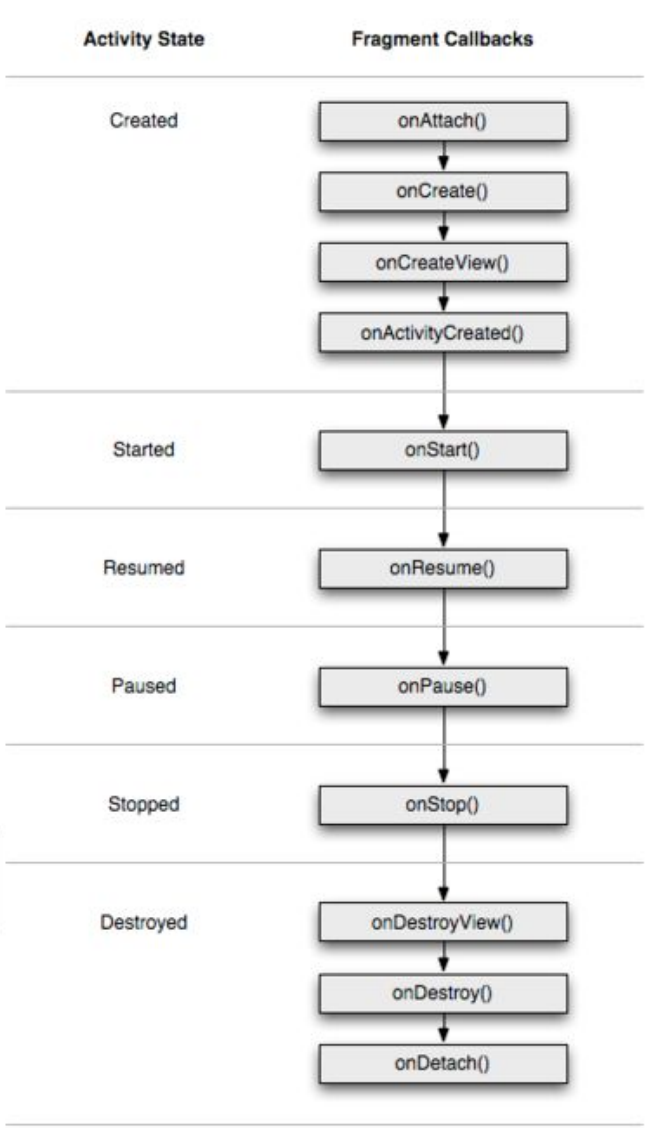
onAttach - onDetach - фрагмент прикреплен к **Activity**

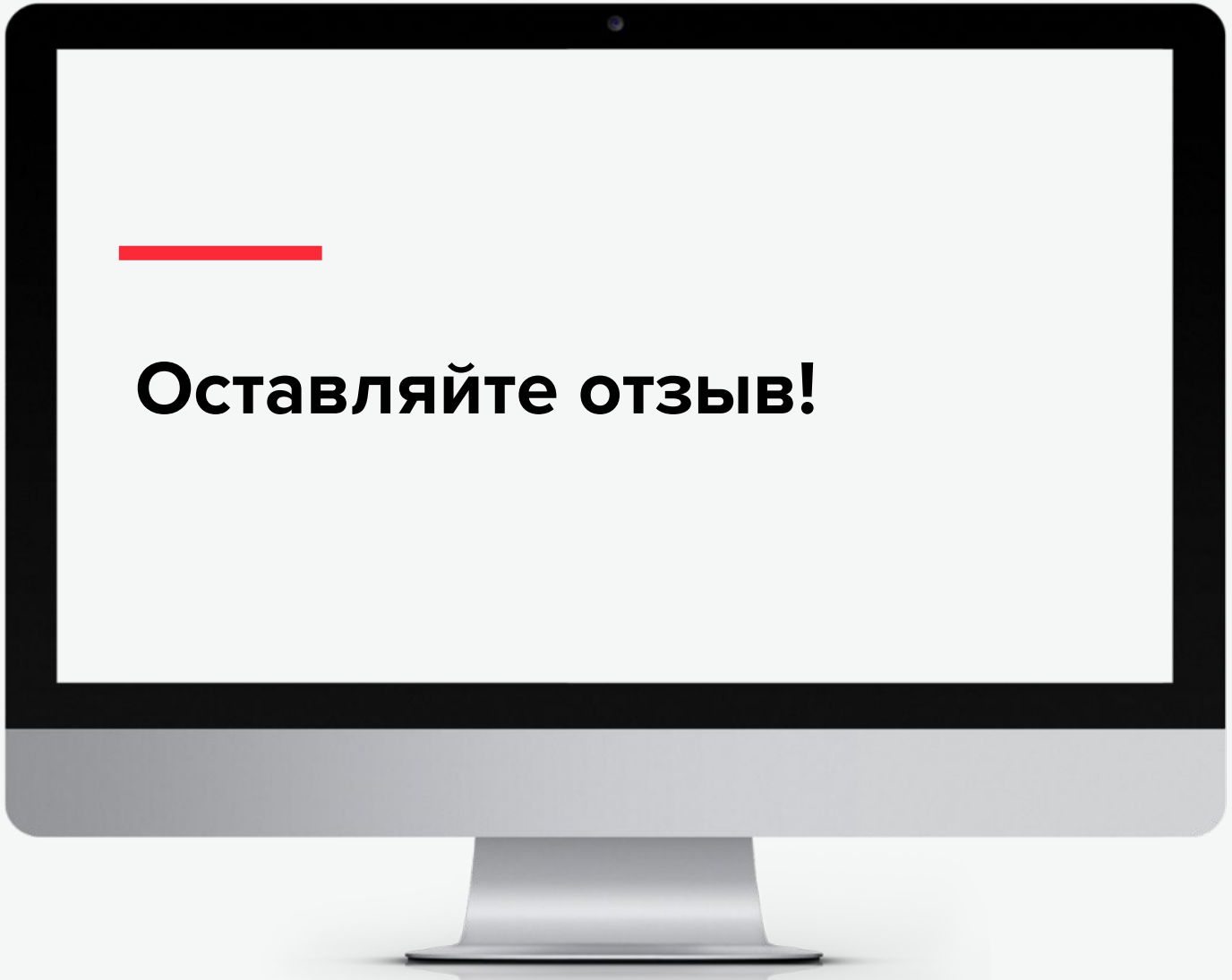
onCreateView - onDestroyView - создать и уничтожить интерфейсную часть

onViewCreated - **View** установлена в **Fragment**

onActivityCreated - у **Activity** точно вызвался метод **onCreate** :)






Оставляйте отзыв!

СПАСИБО
ЗА ВНИМАНИЕ