Ссылка на видео этого руководства на сайте: borisproit.expert

🟢 FOUNDATION (Junior)

1. Что такое Android Framework?

Что такое Android Framework?
What is the Android Framework?

Android Framework — это набор системных API и компонентов, которые предоставляет операционная система Android для разработки приложений.
Он включает классы и механизмы для работы с UI, жизненным циклом компонентов, ресурсами, сетью, базой данных и взаимодействием с системой.

К ключевым компонентам относятся `Activity`, `Fragment`, `Service`, `BroadcastReceiver`, `ContentProvider`, а также `Context`, `Lifecycle` и другие системные классы.

The Android Framework is a set of system APIs and components provided by the Android operating system for building applications.
It includes classes and mechanisms for UI, component lifecycles, resources, networking, databases, and interaction with the system.

Key components include `Activity`, `Fragment`, `Service`, `BroadcastReceiver`, `ContentProvider`, as well as `Context`, `Lifecycle`, and other system classes.

2. **Какие слои входят в архитектуру Android?**

Android имеет многоуровневую архитектуру, состоящую из следующих основных слоёв:

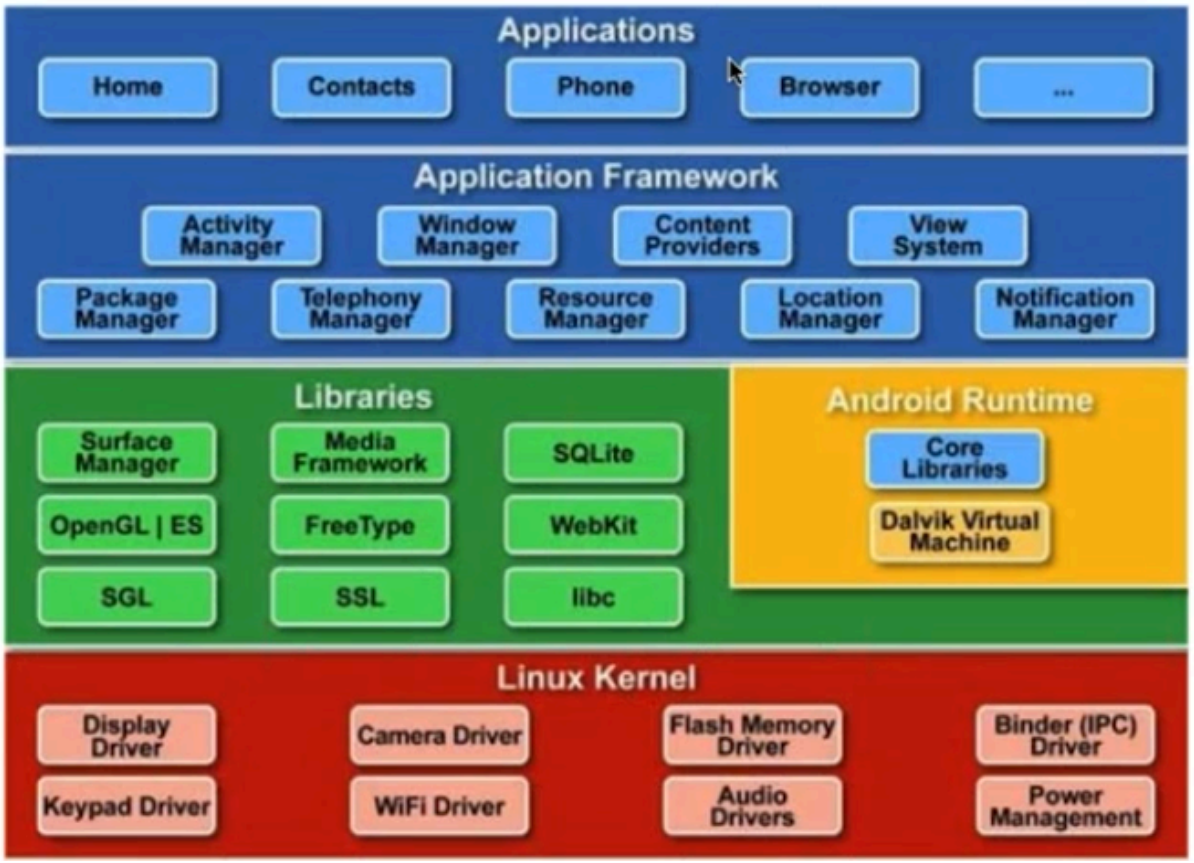
1. **Applications** — приложения пользователя (Phone, Contacts, Browser и т.д.).
2. **Application Framework** — системные сервисы и API для разработки приложений (Activity Manager, Window Manager, View System и др.).
3. **Libraries + Android Runtime (ART)** — нативные библиотеки (SQLite, OpenGL, WebKit) и среда выполнения приложений.
4. **Linux Kernel** — ядро Linux, управляющее драйверами устройств, памятью, процессами и безопасностью.

Ссылка на видео этого руководства на сайте: borisproit.expert

---

Android has a layered architecture consisting of the following main layers:

1. **Applications** — user applications such as Phone, Contacts, and Browser.
2. **Application Framework** — system services and APIs used by apps (Activity Manager, Window Manager, View System, etc.).
3. **Libraries + Android Runtime (ART)** — native libraries (SQLite, OpenGL, WebKit) and the runtime environment for apps.
4. **Linux Kernel** — the Linux kernel that manages device drivers, memory, processes, and security.



3. Что такое Linux Kernel в Android архитектуре?

Что такое Linux Kernel в Android архитектуре?
What is the Linux Kernel in Android architecture?

Ссылка на видео этого руководства на сайте: borisproit.expert

Linux Kernel — это самый нижний слой Android архитектуры.
Он отвечает за взаимодействие с аппаратным обеспечением устройства: управление памятью, процессами, драйверами устройств, сетью и безопасностью.

Android использует модифицированное ядро Linux, которое предоставляет базовые системные функции для всей операционной системы.

The Linux Kernel is the lowest layer in the Android architecture.
It is responsible for interacting with the device hardware, including memory management, process scheduling, device drivers, networking, and security.

Android uses a modified Linux kernel that provides core system services for the entire operating system.

4. Что такое Android Runtime (ART)?

Что такое Android Runtime (ART)?
What is Android Runtime (ART)?

Android Runtime (ART) — это среда выполнения, в которой запускаются приложения Android.
Она отвечает за выполнение байткода приложения, управление памятью и сборку мусора (Garbage Collection).

В ART код приложения компилируется в машинный код (AOT/JIT), что делает запуск и выполнение приложений быстрее.

Android Runtime (ART) is the runtime environment where Android applications execute.
It is responsible for running the app's bytecode, managing memory, and performing garbage collection.

In ART, application code is compiled into machine code (AOT/JIT), which improves startup time and performance.

5. Чем Dalvik отличается от ART?

Чем Dalvik отличается от ART?
What is the difference between Dalvik and ART?

Ссылка на видео этого руководства на сайте: borisproit.expert

Dalvik — это старая виртуальная машина Android, которая выполняла байткод приложений с использованием **JIT (Just-In-Time) компиляции**.
Код компилировался во время выполнения, что могло замедлять запуск и увеличивать нагрузку на CPU.

ART — это новая среда выполнения Android, которая использует **AOT (Ahead-Of-Time) компиляцию**, а также JIT и профилирование.
Код компилируется заранее при установке приложения, поэтому запуск быстрее и производительность выше.

Dalvik was the old Android virtual machine that executed application bytecode using **JIT (Just-In-Time) compilation**.
The code was compiled during execution, which could slow startup and increase CPU usage.

ART is the newer Android runtime that uses **AOT (Ahead-Of-Time) compilation**, along with JIT and profiling.
Code is compiled ahead of time during app installation, resulting in faster startup and better performance.

6. Что такое Android Application Architecture?

Что такое Android Application Architecture?
What is Android Application Architecture?

Android Application Architecture — это структура приложения Android, которая определяет, как организован код, какие слои существуют и как компоненты взаимодействуют между собой.
Её цель — разделить ответственность между слоями, чтобы приложение было масштабируемым, тестируемым и поддерживаемым.

Обычно архитектура приложения делится на три основных слоя:
— **Presentation** — UI и взаимодействие с пользователем (Activity, Fragment, Composable, ViewModel)
— **Domain** — бизнес-логика приложения (UseCases, бизнес-правила)
— **Data** — работа с источниками данных (Repository, API, Database)

Android Application Architecture is the structure used to organize the code of an Android application and define how components interact with each other.
Its goal is to separate responsibilities between layers so the app is scalable, testable, and maintainable.

Ссылка на видео этого руководства на сайте: borisproit.expert

Typically, application architecture consists of three main layers:
— **Presentation** — UI and user interaction (Activity, Fragment, Composable, ViewModel)
— **Domain** — business logic (UseCases, business rules)
— **Data** — data access (Repository, API, database)

🟢 Android components

7. Что такое Activity?

Что такое `Activity`?
What is an `Activity`?

`Activity` — это компонент Android, который представляет **один экран пользовательского интерфейса** и отвечает за взаимодействие пользователя с приложением.
Он управляет жизненным циклом экрана, обрабатывает пользовательские действия и отображает UI.

Каждый экран приложения обычно реализуется как отдельная Activity.

An `Activity` is an Android component that represents **a single screen of the user interface** and handles user interaction with the app.
It manages the screen lifecycle, processes user actions, and displays UI.

Each screen in an Android app is typically implemented as a separate Activity.

```
// Basic Activity example
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main) // sets the UI layout
    }
}
```

8. Что такое Fragment?

Ссылка на видео этого руководства на сайте: borisproit.expert

Что такое `Fragment`?
What is a `Fragment`?

`Fragment` — это компонент Android, который представляет **часть пользовательского интерфейса внутри Activity**.
Он имеет собственный жизненный цикл и может использоваться для создания гибких и переиспользуемых UI-компонентов.

Fragments часто применяются для построения адаптивных интерфейсов (например, на планшетах) и для навигации внутри одной Activity.

A `Fragment` is an Android component that represents **a portion of the user interface within an Activity**.
It has its own lifecycle and is used to build modular and reusable UI components.

Fragments are commonly used for flexible layouts (for example on tablets) and for navigation within a single Activity.

```kotlin
// Basic Fragment example
class UserFragment : Fragment(R.layout.fragment_user) {

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        // UI setup here
    }
}
```

9. Чем Fragment отличается от Activity?

Чем `Fragment` отличается от `Activity`?
What is the difference between a `Fragment` and an `Activity`?

`Activity` — это самостоятельный экран приложения, который является точкой входа для взаимодействия пользователя с системой Android.
Он управляется напрямую Android framework и может существовать независимо.

`Fragment` — это часть интерфейса внутри Activity.
Он не может существовать самостоятельно и всегда должен быть размещён внутри Activity.

An `Activity` is a standalone screen of an Android application and a primary entry point for user interaction with the Android system.
It is managed directly by the Android framework and can exist independently.

Ссылка на видео этого руководства на сайте: borisproit.expert

A `Fragment` is a portion of the UI inside an Activity.
It cannot exist on its own and must always be hosted inside an Activity.

10. Что такое Intent?

Что такое `Intent`?
What is an `Intent`?

`Intent` — это объект Android, который используется для **запроса действия у системы или другого компонента приложения**.
Он применяется для запуска Activity, Service или отправки Broadcast.

Intent может содержать действие, данные и дополнительные параметры (extras).

An `Intent` is an Android object used to **request an action from the system or another application component**.
It is used to start Activities, Services, or send Broadcasts.

An Intent can contain an action, data, and additional parameters (extras).

```
// Starting another Activity using Intent
val intent = Intent(this, DetailActivity::class.java)
intent.putExtra("userId", 42) // passing data
startActivity(intent)
```

11. Чем отличаются explicit и implicit Intent?

Чем отличаются `explicit` и `implicit` Intent?
What is the difference between `explicit` and `implicit` Intent?

Explicit Intent используется, когда приложение точно знает, какой компонент нужно запустить.
В Intent указывается конкретный класс Activity или Service.

Implicit Intent используется, когда приложение описывает действие, которое нужно выполнить,

а Android система сама выбирает подходящий компонент, способный обработать это действие.

An **explicit Intent** is used when the app knows exactly which component should be started. The specific Activity or Service class is specified in the Intent.

An **implicit Intent** is used when the app describes an action to perform, and the Android system finds a suitable component that can handle that action.

```
// Explicit Intent
val explicitIntent = Intent(this, DetailActivity::class.java)
startActivity(explicitIntent)

// Implicit Intent
val implicitIntent = Intent(Intent.ACTION_VIEW)
implicitIntent.data = Uri.parse("https://example.com")
startActivity(implicitIntent)
```

12. Что такое Service?

Что такое `Service`?
What is a `Service`?

`Service` — это компонент Android, предназначенный для выполнения **долгих операций в фоне без пользовательского интерфейса**.
Он используется для задач, которые должны продолжаться, даже если пользователь не взаимодействует с экраном.

Примеры: воспроизведение музыки, загрузка файлов, синхронизация данных.

A `Service` is an Android component used to perform **long-running operations in the background without a user interface**.
It is used for tasks that should continue even when the user is not interacting with the screen.

Examples include music playback, file downloads, and data synchronization.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Basic Service example
class MusicService : Service() {

    override fun onBind(intent: Intent?): IBinder? {
        return null // not a bound service
    }
}
```

13. Что такое BroadcastReceiver?

Что такое `BroadcastReceiver`?
What is a `BroadcastReceiver`?

`BroadcastReceiver` — это компонент Android, который получает и обрабатывает **системные или пользовательские broadcast-сообщения**.
Он используется для реакции на события системы, например: изменение сети, низкий заряд батареи или завершение загрузки устройства.

Receiver не имеет пользовательского интерфейса и обычно выполняет короткую работу при получении события.

A `BroadcastReceiver` is an Android component that listens for and responds to **system or application broadcast messages**.
It is used to react to system events such as network changes, low battery, or device boot completion.

A receiver has no UI and typically performs a short task when the event is received.

```
// BroadcastReceiver example
class NetworkReceiver : BroadcastReceiver() {

    override fun onReceive(context: Context, intent: Intent) {
        // handle broadcast event
    }
}
```

14. Что такое ContentProvider?

Ссылка на видео этого руководства на сайте: borisproit.expert

Что такое `ContentProvider`?
What is a `ContentProvider`?

`ContentProvider` — это компонент Android, предназначенный для **управления и предоставления доступа к данным приложения** другим приложениям или компонентам.
Он обеспечивает безопасный и стандартизированный способ чтения и записи данных через `ContentResolver`.

Чаще всего используется для доступа к данным контактов, медиафайлов или собственной базы данных приложения.

A `ContentProvider` is an Android component used to **manage and share application data** with other apps or components.
It provides a secure and standardized way to read and write data through the `ContentResolver`.

It is commonly used for accessing contacts, media files, or an app's own database.

```
// ContentProvider skeleton
class UserProvider : ContentProvider() {

    override fun query(
        uri: Uri,
        projection: Array<out String>?,
        selection: String?,
        selectionArgs: Array<out String>?,
        sortOrder: String?
    ): Cursor? {
        return null // return data cursor
    }
}
```

🟢 Context

15. Что такое Context?

Ссылка на видео этого руководства на сайте: borisproit.expert

Что такое `Context`?
What is `Context`?

`Context` — это объект Android, который предоставляет доступ к системным ресурсам и сервисам приложения.
Через Context можно получать ресурсы, запускать Activity, работать с файлами, базой данных и системными сервисами.

Он является связующим звеном между приложением и Android системой.

`Context` is an Android object that provides access to application resources and system services.
Through a Context you can access resources, start Activities, work with files, databases, and system services.

It acts as a bridge between the application and the Android system.

```
// Using Context to start an Activity
val intent = Intent(context, DetailActivity::class.java)
context.startActivity(intent)
```

16. Какие типы Context существуют?

Какие типы `Context` существуют?
What types of `Context` exist?

Основные типы Context в Android:

1. **Application Context** — живёт столько же, сколько всё приложение.
   Используется для операций, не привязанных к UI (например, база данных, DI, системные сервисы).

2. **Activity Context** — привязан к жизненному циклу Activity.
   Используется для операций, связанных с UI (запуск Activity, отображение диалогов).

3. **Service Context** — Context внутри Service.
   Используется для фоновых операций.

The main types of Context in Android are:

Ссылка на видео этого руководства на сайте: borisproit.expert

1. **Application Context** — lives as long as the entire application.
   Used for operations not tied to UI (for example databases, DI, system services).

2. **Activity Context** — tied to the lifecycle of an Activity.
   Used for UI-related operations (starting Activities, showing dialogs).

3. **Service Context** — the Context inside a Service.
   Used for background operations.

🟢 Android project structure

18. Что такое AndroidManifest?

Что такое `AndroidManifest`?
What is `AndroidManifest`?

`AndroidManifest.xml` — это файл конфигурации Android-приложения, в котором описываются основные компоненты приложения и его настройки.
Он сообщает системе Android, какие компоненты есть в приложении и какие разрешения ему нужны.

В Manifest обычно объявляются `Activity`, `Service`, `BroadcastReceiver`, `ContentProvider`, а также permissions и точка входа приложения.

`AndroidManifest.xml` is the configuration file of an Android application that describes the app's main components and settings.
It tells the Android system what components the app contains and what permissions it requires.

The Manifest typically declares `Activity`, `Service`, `BroadcastReceiver`, `ContentProvider`, as well as permissions and the app entry point.

Ссылка на видео этого руководства на сайте: borisproit.expert

```xml
<!-- AndroidManifest.xml example -->
<manifest package="com.example.app">

    <application
        android:name=".MyApp">

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

    </application>

</manifest>
```

19. Что находится в папке **res**?

Что находится в папке `res`?
What is in the `res` folder?

Папка `res` содержит **ресурсы приложения**, которые используются в UI и конфигурации.
Ресурсы отделены от кода, чтобы их можно было легко менять, локализовать и переиспользовать.

Основные типы ресурсов:

— `layout` — XML-файлы интерфейса
— `drawable` — изображения и графика
— `values` — строки, цвета, стили, размеры
— `mipmap` — иконки приложения
— `menu` — меню приложения
— `anim` / `animator` — анимации

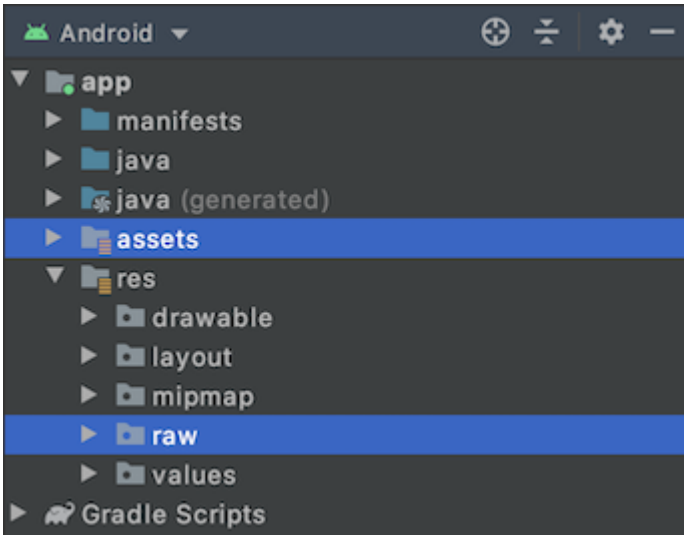
The `res` folder contains **application resources** used by the UI and configuration.
Resources are separated from code so they can be easily changed, localized, and reused.

Common resource types include:

— `layout` — UI XML layouts
— `drawable` — images and graphics

Ссылка на видео этого руководства на сайте: borisproit.expert

— `values` — strings, colors, styles, dimensions
— `mipmap` — app icons
— `menu` — application menus
— `anim` / `animator` — animations



20. Что находится в папке **src**?

Что находится в папке `src`?
What is in the `src` folder?

Папка `src` содержит **исходный код приложения**.
Здесь размещаются Kotlin/Java классы, тесты и код, который реализует логику приложения.

В Android-проекте обычно есть несколько подпапок:

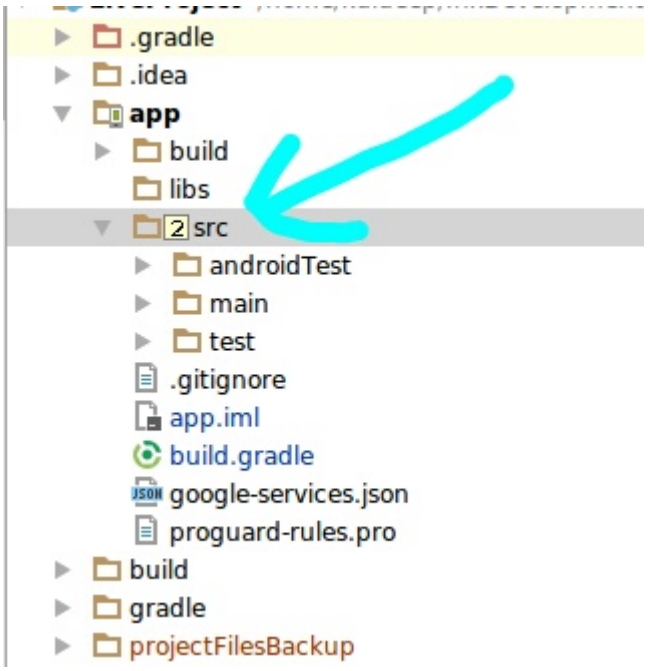
— `main` — основной код приложения
— `test` — unit-тесты (запускаются на JVM)
— `androidTest` — инструментальные тесты (запускаются на устройстве или эмуляторе)

The `src` folder contains the **source code of the application**.
It includes Kotlin/Java classes, tests, and the code that implements the app logic.

In an Android project it typically contains several subfolders:

— `main` — main application code
— `test` — unit tests (run on the JVM)
— `androidTest` — instrumentation tests (run on a device or emulator)

Ссылка на видео этого руководства на сайте: borisproit.expert



21. Что находится в папке **assets**?

Что находится в папке `assets`?
What is in the `assets` folder?

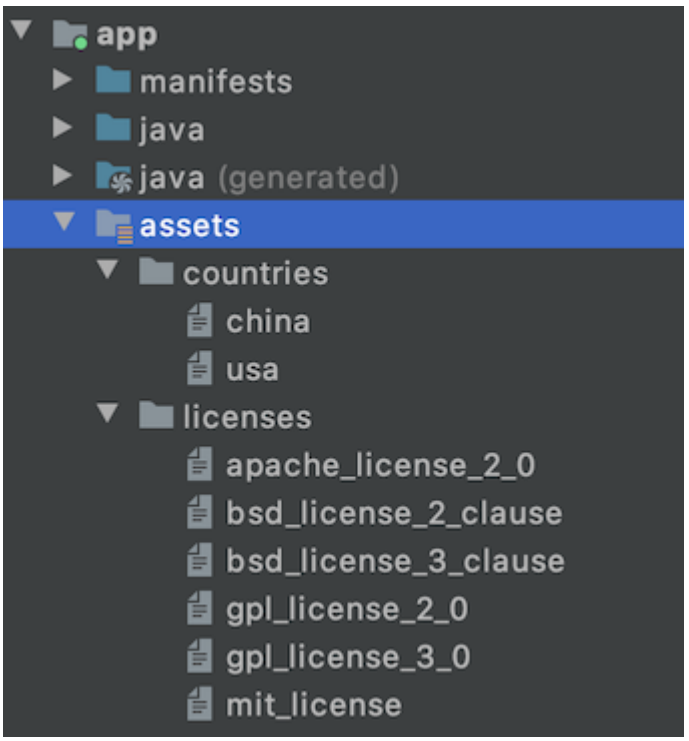
Папка `assets` содержит **произвольные файлы приложения**, которые нужно включить в APK без обработки системой Android.
Эти файлы сохраняются в оригинальном виде и доступны приложению во время выполнения.

Чаще всего там хранятся JSON, HTML, шрифты, модели ML, большие текстовые файлы и другие ресурсы, которые не подходят для папки `res`.

The `assets` folder contains **raw application files** that are packaged into the APK without being processed by the Android build system.
These files are kept in their original form and can be accessed at runtime.

It is commonly used for JSON files, HTML files, fonts, ML models, large text files, and other resources that do not fit into the `res` folder.

Ссылка на видео этого руководства на сайте: borisproit.expert



22. Что находится в папке **values**?

Что находится в папке `values`?
What is in the `values` folder?

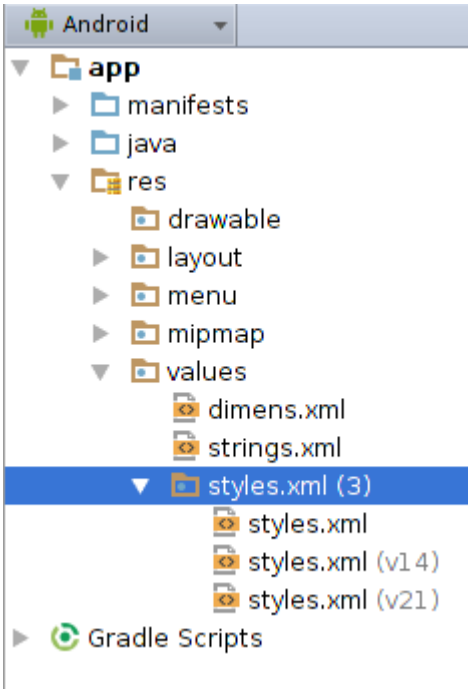
Папка `values` содержит **простые ресурсы приложения**, описанные в XML.
Здесь обычно хранятся строки, цвета, стили, темы, размеры и другие конфигурационные значения.

Это позволяет централизованно управлять ресурсами и легко поддерживать локализацию и переиспользование.

The `values` folder contains **simple application resources** defined in XML.
It typically stores strings, colors, styles, themes, dimensions, and other configuration values.

This allows centralized resource management and makes localization and reuse easier.

Ссылка на видео этого руководства на сайте: borisproit.expert



🟢 Build / APK

23. Что такое APK файл?

Что такое APK файл?
What is an APK file?

APK (Android Package) — это установочный файл Android-приложения.
Он содержит весь код, ресурсы и метаданные, необходимые для установки и запуска приложения на устройстве.

APK включает compiled code, ресурсы, AndroidManifest и другие файлы, которые Android использует для установки приложения.

An APK (Android Package) is the installation file for an Android application.
It contains all the code, resources, and metadata required to install and run the app on a device.

An APK includes compiled code, resources, the AndroidManifest, and other files used by Android during installation.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Android Studio builds the APK when you run:
Build -> Build APK
```

24. Какие файлы входят в APK?

Какие файлы входят в APK?
What files are included in an APK?

APK — это архив (по сути ZIP), который содержит все файлы приложения, необходимые для установки и запуска.

Основные части APK:

— `classes.dex` — скомпилированный байткод приложения (Dex format)
— `AndroidManifest.xml` — описание компонентов приложения и разрешений
— `res/` — ресурсы приложения (layout, drawable, values и др.)
— `assets/` — необработанные файлы приложения
— `lib/` — нативные библиотеки (`.so`) для разных архитектур CPU
— `META-INF/` — информация о подписи приложения
— `resources.arsc` — скомпилированная таблица ресурсов

An APK is essentially a ZIP archive containing all files required to install and run an Android application.

Main contents of an APK:

— `classes.dex` — compiled application bytecode (Dex format)
— `AndroidManifest.xml` — app component and permission declarations
— `res/` — application resources (layout, drawable, values, etc.)
— `assets/` — raw application files
— `lib/` — native libraries (`.so`) for different CPU architectures
— `META-INF/` — application signing information
— `resources.arsc` — compiled resource table

🟢 Android tools

Ссылка на видео этого руководства на сайте: borisproit.expert

25. Что такое Android SDK?

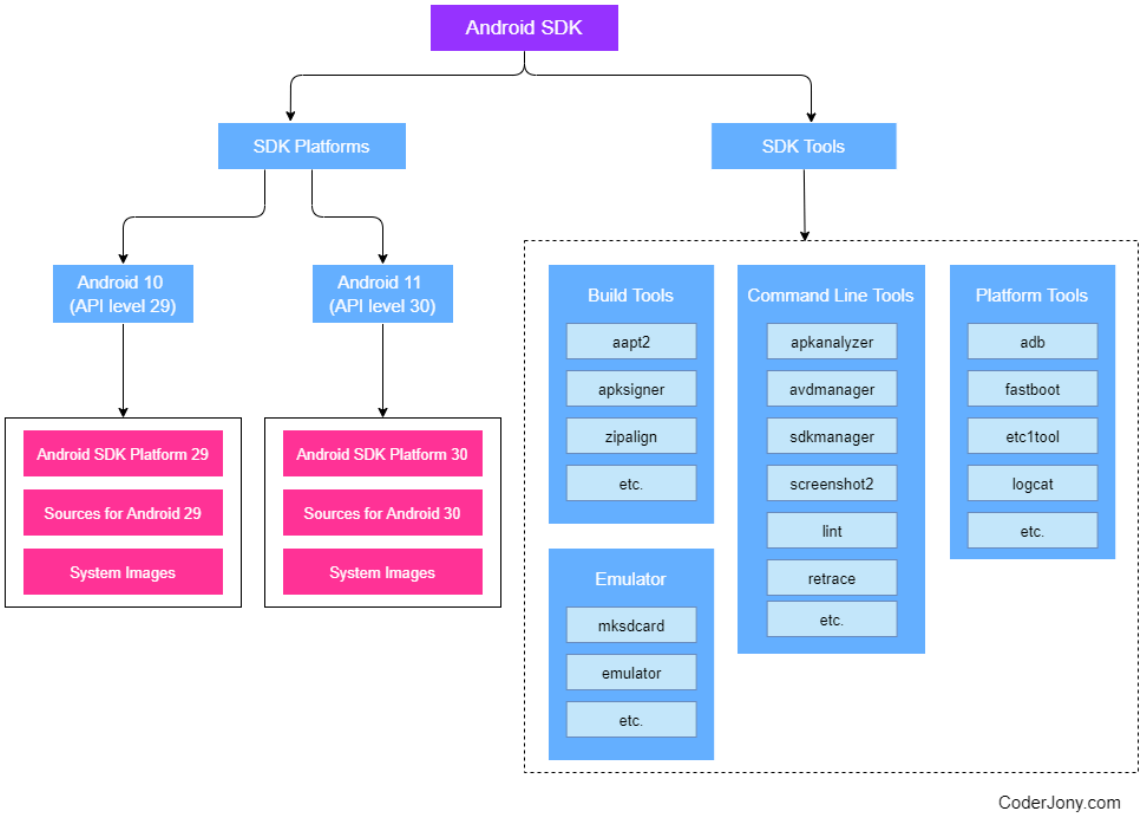
Что такое `Android SDK`?
What is the `Android SDK`?

`Android SDK` (Software Development Kit) — это набор инструментов, библиотек и API, который используется для разработки Android-приложений.
Он предоставляет всё необходимое для создания, сборки, тестирования и отладки приложений.

В SDK входят Android API, инструменты сборки, эмулятор и другие утилиты для разработки.

The `Android SDK` (Software Development Kit) is a collection of tools, libraries, and APIs used to develop Android applications.
It provides everything needed to build, test, and debug Android apps.

The SDK includes Android APIs, build tools, the emulator, and other development utilities.



Android SDK Classification

26. Какие инструменты входят в Android SDK?

Ссылка на видео этого руководства на сайте: borisproit.expert

Какие инструменты входят в `Android SDK`?
What tools are included in the `Android SDK`?

Android SDK включает набор инструментов, которые используются для разработки, сборки, тестирования и отладки Android-приложений.

Основные инструменты:

— **Android SDK Platform Tools** — базовые инструменты, включая `adb` (Android Debug Bridge) для взаимодействия с устройством.
— **Android SDK Build Tools** — инструменты сборки приложения (`aapt`, `d8`, `zipalign`, `apksigner`).
— **Android Emulator** — эмулятор Android-устройства для тестирования приложений.
— **SDK Manager** — инструмент для установки и обновления компонентов SDK.
— **AVD Manager** — инструмент для создания и управления виртуальными устройствами (Android Virtual Devices).
— **Lint** — инструмент статического анализа кода.

The Android SDK includes a set of tools used for developing, building, testing, and debugging Android applications.

Main tools include:

— **Android SDK Platform Tools** — core tools including `adb` (Android Debug Bridge) for communicating with devices.
— **Android SDK Build Tools** — tools used to build applications (`aapt`, `d8`, `zipalign`, `apksigner`).
— **Android Emulator** — a virtual Android device used for testing apps.
— **SDK Manager** — tool for installing and updating SDK components.
— **AVD Manager** — tool for creating and managing Android Virtual Devices.
— **Lint** — static code analysis tool.

27. Что такое ADB?

Что такое `ADB`?
What is `ADB`?

`ADB` (Android Debug Bridge) — это инструмент командной строки из Android SDK, который позволяет разработчику взаимодействовать с Android-устройством или эмулятором.
Он используется для установки приложений, просмотра логов, выполнения команд на устройстве и отладки.

Ссылка на видео этого руководства на сайте: borisproit.expert

---

ADB (Android Debug Bridge) is a command-line tool from the Android SDK that allows developers to communicate with an Android device or emulator.
It is used to install apps, view logs, run commands on the device, and perform debugging.

```
// Install APK on device
adb install app.apk

// View connected devices
adb devices

// Show device logs
adb logcat
```

28. Что такое Android Emulator?

Что такое Android Emulator?
What is the Android Emulator?

Android Emulator — это инструмент из Android SDK, который позволяет запускать виртуальное Android-устройство на компьютере.
Он используется для тестирования и отладки приложений без необходимости использовать физическое устройство.

Эмулятор имитирует поведение реального устройства: экран, сенсор, сеть, GPS, камеру и другие аппаратные функции.

The **Android Emulator** is a tool from the Android SDK that allows developers to run a virtual Android device on a computer.
It is used to test and debug applications without needing a physical device.

The emulator simulates real device behavior such as the screen, touch input, network, GPS, camera, and other hardware features.

🟢 Data transfer

Ссылка на видео этого руководства на сайте: borisproit.expert

29. Что такое Bundle?

Что такое `Bundle`?
What is a `Bundle`?

`Bundle` — это контейнер для хранения **пар ключ-значение**, который используется для передачи данных между компонентами Android.
Он часто применяется для передачи данных через `Intent`, сохранения состояния Activity или передачи аргументов во Fragment.

`Bundle` поддерживает примитивные типы данных и объекты, реализующие `Parcelable` или `Serializable`.

A `Bundle` is a container used to store **key-value pairs** for passing data between Android components.
It is commonly used with `Intent`, for saving Activity state, or for passing arguments to a Fragment.

A `Bundle` supports primitive data types and objects implementing `Parcelable` or `Serializable`.

```
// Putting data into Bundle
val bundle = Bundle()
bundle.putString("name", "Alex")

// Reading data
val name = bundle.getString("name")
```

30. Как передать данные между Activity?

Как передать данные между `Activity`?
How do you pass data between `Activity`?

Данные между Activity передаются через `Intent` с использованием `extras`.
Одна Activity добавляет данные в Intent, а другая читает их после запуска.

Data between Activities is passed using an `Intent` with `extras`.
One Activity puts the data into the Intent, and the receiving Activity reads it after being started.

Ссылка на видео этого руководства на сайте: borisproit.expert

```kotlin
// Sending data
val intent = Intent(this, DetailActivity::class.java)
intent.putExtra("userId", 42) // add data
startActivity(intent)

// Receiving data
val userId = intent.getIntExtra("userId", -1) // read data
```

🟢 Permissions

31. Что такое permissions в Android?

Что такое `permissions` в Android?
What are `permissions` in Android?

`Permissions` — это механизм безопасности Android, который контролирует доступ приложения к чувствительным данным и системным функциям устройства.
Например: доступ к камере, геолокации, контактам или хранилищу.

Разрешения объявляются в `AndroidManifest.xml`, а некоторые из них требуют подтверждения пользователя во время работы приложения (runtime permissions).

`Permissions` are a security mechanism in Android that controls an app's access to sensitive data and system features of the device.
For example: access to the camera, location, contacts, or storage.

Permissions are declared in `AndroidManifest.xml`, and some of them must also be granted by the user at runtime (runtime permissions).

Ссылка на видео этого руководства на сайте: borisproit.expert

XML

```xml
<!-- Declaring permission in AndroidManifest -->
<uses-permission android:name="android.permission.CAMERA"/>
```

Kotlin

```kotlin
// Requesting permission at runtime
ActivityCompat.requestPermissions(
    this,
    arrayOf(Manifest.permission.CAMERA),
    100
)
```

32. Что такое runtime permissions?

Что такое `runtime permissions`?
What are `runtime permissions`?

`Runtime permissions` — это разрешения, которые приложение должно **запрашивать у пользователя во время работы**, а не только объявлять в `AndroidManifest`.
Пользователь видит диалог и может разрешить или отклонить доступ.

Этот механизм был введён начиная с **Android 6.0 (API 23)** для повышения безопасности.

`Runtime permissions` are permissions that an app must **request from the user while the app is running**, not just declare in the `AndroidManifest`.
The user sees a dialog and can allow or deny the permission.

This mechanism was introduced starting from **Android 6.0 (API 23)** to improve security.

```kotlin
// Request camera permission at runtime
ActivityCompat.requestPermissions(
    this,
    arrayOf(Manifest.permission.CAMERA),
    100
)
```

Ссылка на видео этого руководства на сайте: borisproit.expert

🟢 Activity lifecycle

33. Какие методы есть в lifecycle Activity?

Какие методы есть в `lifecycle` Activity?
What methods exist in the Activity lifecycle?

Lifecycle Activity состоит из набора методов, которые вызываются системой Android при изменении состояния Activity.
Основные методы жизненного цикла:

— `onCreate()` — вызывается при создании Activity
— `onStart()` — Activity становится видимой
— `onResume()` — Activity выходит на передний план и готова к взаимодействию
— `onPause()` — Activity частично теряет фокус
— `onStop()` — Activity больше не видна
— `onDestroy()` — Activity уничтожается
— `onRestart()` — Activity перезапускается после `onStop()`

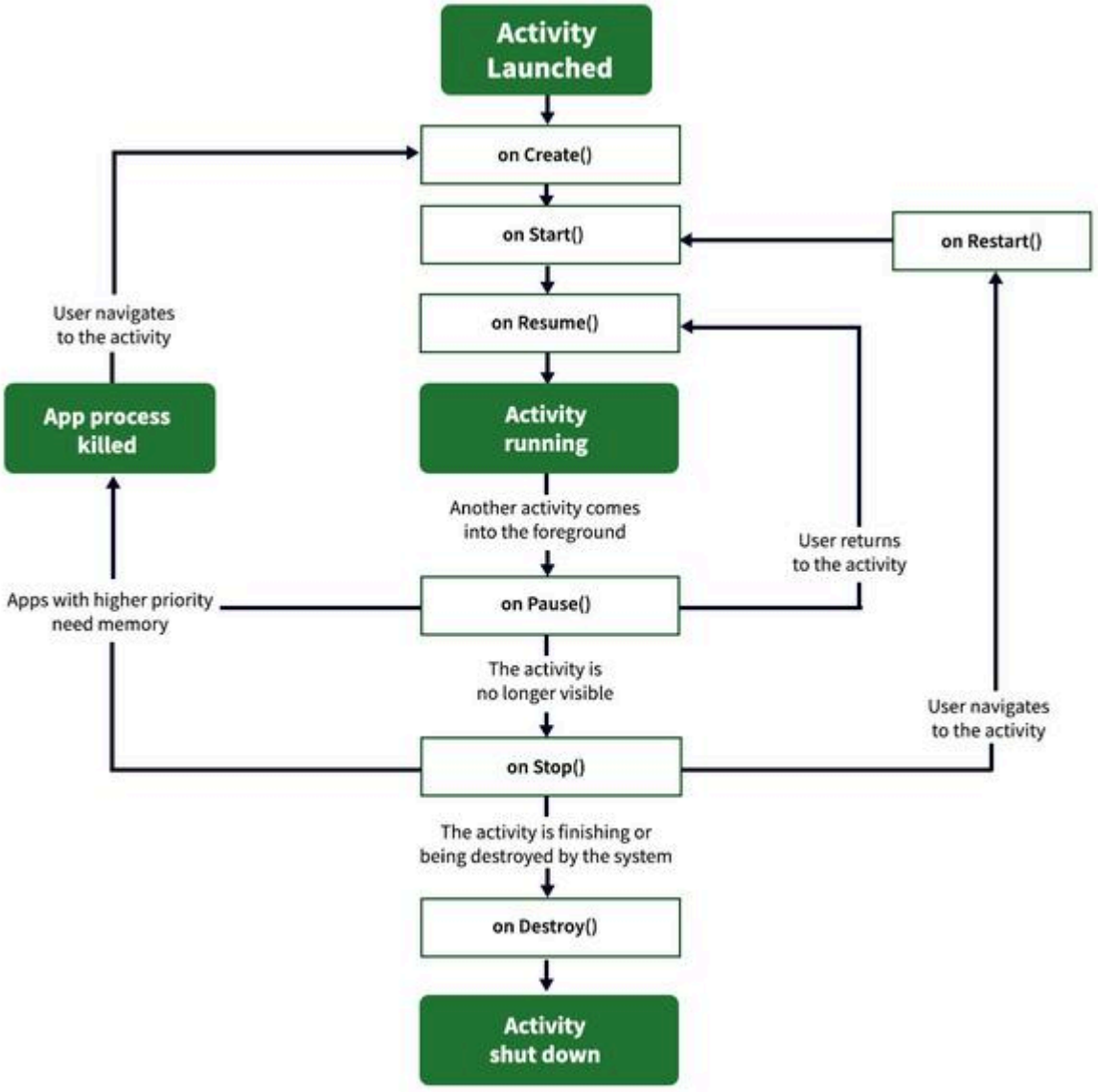
The Activity lifecycle consists of methods that are called by the Android system when the Activity state changes.
The main lifecycle methods are:

— `onCreate()` — called when the Activity is created
— `onStart()` — the Activity becomes visible
— `onResume()` — the Activity moves to the foreground and is ready for interaction
— `onPause()` — the Activity partially loses focus
— `onStop()` — the Activity is no longer visible
— `onDestroy()` — the Activity is destroyed
— `onRestart()` — the Activity restarts after being stopped

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Activity lifecycle example
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }

    override fun onStart() { super.onStart() }

    override fun onResume() { super.onResume() }

    override fun onPause() { super.onPause() }

    override fun onStop() { super.onStop() }

    override fun onDestroy() { super.onDestroy() }
}
```

Ссылка на видео этого руководства на сайте: borisproit.expert



34. Когда вызывается onCreate()?

Когда вызывается `onCreate()`?
When is `onCreate()` called?

`onCreate()` вызывается, когда Activity **создаётся впервые**.
Это первый lifecycle-метод Activity, где обычно инициализируется UI, выполняется настройка зависимостей и восстанавливается состояние.

Он вызывается один раз при создании Activity.

Ссылка на видео этого руководства на сайте: borisproit.expert

---

`onCreate()` is called when an Activity is **created for the first time**.
It is the first lifecycle method where UI initialization, dependency setup, and state restoration usually happen.

It is typically called once during the Activity creation.

```
// onCreate example
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main) // initialize UI
}
```

35. Почему setContentView() обычно вызывается в onCreate()?

Почему `setContentView()` обычно вызывается в `onCreate()`?
Why is `setContentView()` usually called in `onCreate()`?

`setContentView()` вызывается в `onCreate()`, потому что в этот момент Activity только создаётся и нужно инициализировать пользовательский интерфейс.
Этот метод устанавливает layout, который станет содержимым экрана Activity.

После этого можно безопасно находить View (`findViewById`) и настраивать UI.

`setContentView()` is usually called in `onCreate()` because this is when the Activity is first created and the user interface needs to be initialized.
This method sets the layout that will become the Activity's screen content.

After this call, it is safe to access Views (`findViewById`) and configure the UI.

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

    setContentView(R.layout.activity_main) // sets the layout

    val button = findViewById<Button>(R.id.myButton) // safe after layout is set
}
```

36. Когда вызывается onStart()?

Ссылка на видео этого руководства на сайте: borisproit.expert

Когда вызывается `onStart()`?
When is `onStart()` called?

`onStart()` вызывается, когда Activity **становится видимой пользователю**, но ещё не находится на переднем плане для взаимодействия.
Он вызывается после `onCreate()` или после `onRestart()`.

`onStart()` is called when the Activity **becomes visible to the user**, but is not yet in the foreground for interaction.
It is called after `onCreate()` or after `onRestart()`.

```
override fun onStart() {
    super.onStart()
    // Activity is now visible
}
```

37. Когда вызывается onResume()?

Когда вызывается `onResume()`?
When is `onResume()` called?

`onResume()` вызывается, когда Activity **переходит на передний план и готова к взаимодействию с пользователем**.
В этот момент Activity полностью активна и получает пользовательский ввод.

Метод вызывается после `onStart()` или когда Activity возвращается из состояния `onPause()`.

`onResume()` is called when the Activity **comes to the foreground and is ready for user interaction**.
At this point the Activity is fully active and can receive user input.

This method is called after `onStart()` or when the Activity returns from the `onPause()` state.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
override fun onResume() {
    super.onResume()
    // Activity is now in the foreground and interactive
}
```

38. Когда вызывается onPause()?

Когда вызывается `onPause()`?
When is `onPause()` called?

`onPause()` вызывается, когда Activity **частично теряет фокус**, потому что другая Activity начинает появляться поверх неё.
В этот момент Activity всё ещё может быть частично видна, но пользователь уже не может с ней взаимодействовать.

Этот метод используется для сохранения состояния, остановки анимаций или освобождения ресурсов.

`onPause()` is called when the Activity **partially loses focus**, because another Activity is starting to appear on top of it.
At this point the Activity may still be partially visible, but the user can no longer interact with it.

This method is commonly used to save state, pause animations, or release resources.

```
override fun onPause() {
    super.onPause()
    // pause ongoing work (e.g., animations, sensors)
}
```

39. Когда вызывается onStop()?

Когда вызывается `onStop()`?
When is `onStop()` called?

`onStop()` вызывается, когда Activity **полностью перестаёт быть видимой пользователю**.
Это происходит, когда другая Activity полностью закрывает текущую или приложение уходит в фон.

В этом методе обычно освобождают ресурсы, которые не нужны, пока Activity не видна.

Ссылка на видео этого руководства на сайте: borisproit.expert

---

`onStop()` is called when the Activity **is no longer visible to the user**.
This happens when another Activity completely covers it or when the app goes to the background.

This method is typically used to release resources that are not needed while the Activity is not visible.

```
override fun onStop() {
    super.onStop()
    // release resources not needed while Activity is hidden
}
```

40. Когда вызывается onDestroy()?

Когда вызывается `onDestroy()`?
When is `onDestroy()` called?

`onDestroy()` вызывается, когда Activity **уничтожается системой**.
Это происходит либо когда Activity завершает работу (`finish()`), либо когда система уничтожает её для освобождения ресурсов.

Метод используется для окончательной очистки ресурсов.

`onDestroy()` is called when the Activity **is being destroyed by the system**.
This can happen when the Activity finishes (`finish()`) or when the system destroys it to reclaim resources.

This method is typically used for final cleanup.

```
override fun onDestroy() {
    super.onDestroy()
    // final cleanup
}
```

41. Когда вызывается onRestart()?

Когда вызывается `onRestart()`?
When is `onRestart()` called?

Ссылка на видео этого руководства на сайте: borisproit.expert

`onRestart()` вызывается, когда Activity **возвращается из состояния `onStop()` и снова готовится стать видимой**.
После `onRestart()` обычно вызывается `onStart()`.

`onRestart()` is called when an Activity **is coming back from the `onStop()` state and is about to become visible again**.
After `onRestart()`, the `onStart()` method is typically called.

```
override fun onRestart() {
    super.onRestart()
    // Activity is restarting after being stopped
}
```

🟢 Fragment lifecycle

42. Какие методы есть в lifecycle Fragment?

Какие методы есть в `lifecycle` Fragment?
What methods exist in the Fragment lifecycle?

Lifecycle Fragment включает несколько методов, которые вызываются системой Android при создании, отображении и уничтожении Fragment.

Основные методы:

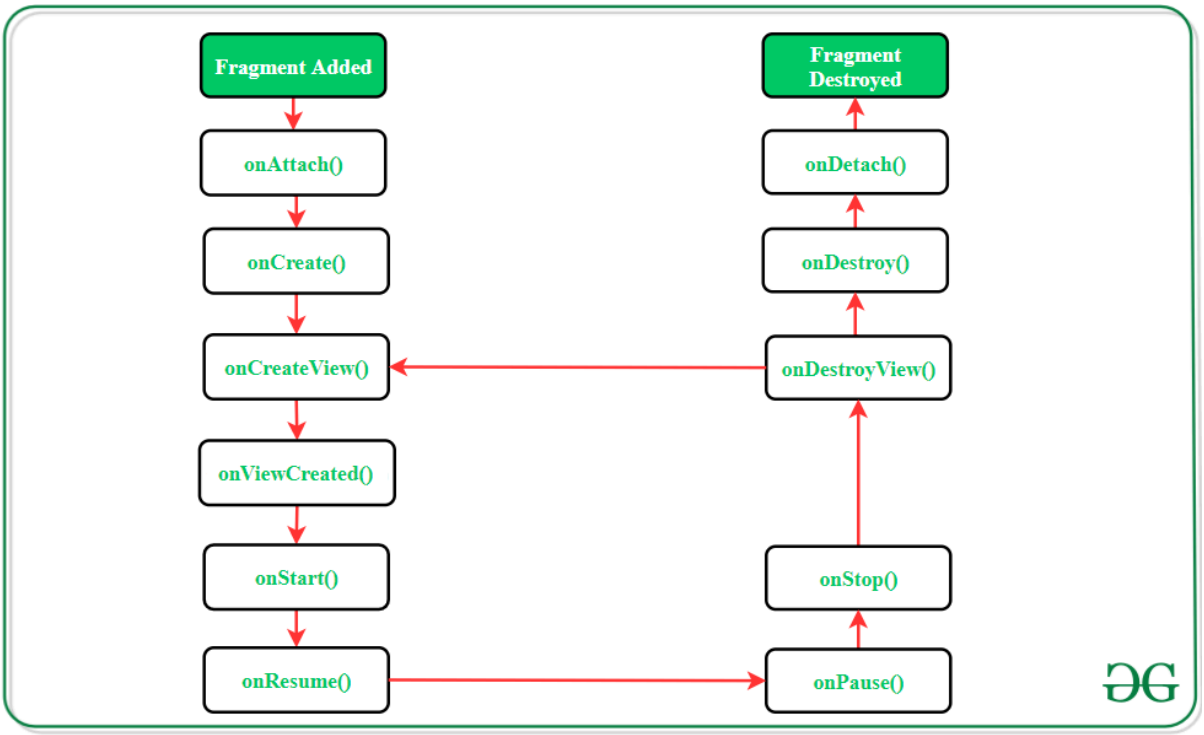
— `onAttach()` — Fragment прикрепляется к Activity
— `onCreate()` — Fragment создаётся
— `onCreateView()` — создаётся UI Fragment
— `onViewCreated()` — View уже создана и можно настраивать UI
— `onStart()` — Fragment становится видимым
— `onResume()` — Fragment активен и готов к взаимодействию
— `onPause()` — Fragment теряет фокус
— `onStop()` — Fragment больше не виден
— `onDestroyView()` — уничтожается View Fragment
— `onDestroy()` — Fragment уничтожается
— `onDetach()` — Fragment отсоединяется от Activity

Ссылка на видео этого руководства на сайте: borisproit.expert

The Fragment lifecycle consists of several methods called by the Android system when a Fragment is created, displayed, and destroyed.

Main lifecycle methods:

— `onAttach()` — Fragment is attached to the Activity
— `onCreate()` — Fragment is created
— `onCreateView()` — Fragment UI is created
— `onViewCreated()` — View is created and UI setup can begin
— `onStart()` — Fragment becomes visible
— `onResume()` — Fragment is active and ready for interaction
— `onPause()` — Fragment loses focus
— `onStop()` — Fragment is no longer visible
— `onDestroyView()` — Fragment view is destroyed
— `onDestroy()` — Fragment is destroyed
— `onDetach()` — Fragment is detached from the Activity



43. Чем lifecycle Fragment отличается от Activity lifecycle?

Чем lifecycle `Fragment` отличается от lifecycle `Activity`?
What is the difference between the Fragment lifecycle and the Activity lifecycle?

Lifecycle Fragment **более сложный**, потому что Fragment зависит от Activity и имеет **два жизненных цикла**:
— lifecycle самого Fragment
— lifecycle его View

Ссылка на видео этого руководства на сайте: borisproit.expert

View Fragment может быть уничтожена (`onDestroyView()`), при этом сам Fragment остаётся в памяти.

Activity же имеет **один lifecycle**, связанный с жизнью всего экрана.

The Fragment lifecycle is **more complex** because a Fragment depends on an Activity and has **two lifecycles**:
— the lifecycle of the Fragment itself
— the lifecycle of its View

The Fragment's View can be destroyed (`onDestroyView()`), while the Fragment instance may still remain in memory.

An Activity, on the other hand, has **a single lifecycle** tied to the lifetime of the screen.

44. Когда вызывается onAttach()?

Когда вызывается `onAttach()`?
When is `onAttach()` called?

`onAttach()` вызывается, когда Fragment **впервые прикрепляется к Activity**.
В этот момент Fragment получает `Context` и может начать взаимодействовать с Activity.

Это первый метод lifecycle Fragment.

`onAttach()` is called when a Fragment **is first attached to an Activity**.
At this point the Fragment receives a `Context` and can start interacting with the Activity.

It is the first method in the Fragment lifecycle.

```
override fun onAttach(context: Context) {
    super.onAttach(context)
    // Fragment is now attached to Activity
}
```

45. Когда вызывается onCreateView()?

Ссылка на видео этого руководства на сайте: borisproit.expert

Когда вызывается `onCreateView()`?
When is `onCreateView()` called?

`onCreateView()` вызывается, когда Fragment **должен создать свой пользовательский интерфейс**.
В этом методе обычно создаётся и возвращается View, которая станет UI Fragment.

Метод вызывается после `onCreate()` и перед `onViewCreated()`.

`onCreateView()` is called when the Fragment **needs to create its user interface**.
In this method, the View that represents the Fragment UI is usually created and returned.

It is called after `onCreate()` and before `onViewCreated()`.

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

46. Когда вызывается onViewCreated()?

Когда вызывается `onViewCreated()`?
When is `onViewCreated()` called?

`onViewCreated()` вызывается **сразу после того, как View Fragment была создана** в `onCreateView()`.
В этом методе обычно настраивают UI: инициализируют View, устанавливают listeners и подписываются на данные.

`onViewCreated()` is called **right after the Fragment's View has been created** in `onCreateView()`.
This method is typically used to configure the UI: initialize Views, set listeners, and observe data.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    val button = view.findViewById<Button>(R.id.myButton)
    button.setOnClickListener {
        // handle click
    }
}
```

47. Когда вызывается onActivityCreated()?

Когда вызывается `onActivityCreated()`?
When is `onActivityCreated()` called?

`onActivityCreated()` вызывался после того, как **Activity и её `onCreate()` полностью завершились**, и Fragment уже был прикреплён к Activity.
Этот метод использовался для инициализации логики, которая зависела от Activity.

⚠️ Начиная с **Android API 28**, этот метод **deprecated**.
Вместо него рекомендуется использовать `onViewCreated()` или `onCreate()`.

`onActivityCreated()` was called after the **Activity finished its `onCreate()`**, and the Fragment was fully attached to it.
It was used for initialization that depended on the Activity.

⚠️ Starting from **Android API 28**, this method is **deprecated**.
You should use `onViewCreated()` or `onCreate()` instead.

```
@Deprecated("Use onViewCreated instead")
override fun onActivityCreated(savedInstanceState: Bundle?) {
    super.onActivityCreated(savedInstanceState)
}
```

48. Когда вызывается onDestroyView()?

Когда вызывается `onDestroyView()`?
When is `onDestroyView()` called?

`onDestroyView()` вызывается, когда **View Fragment уничтожается**, но сам Fragment ещё может оставаться в памяти.

Ссылка на видео этого руководства на сайте: borisproit.expert

Это происходит, например, когда Fragment помещается в back stack или его UI пересоздаётся.

В этом методе обычно очищают ссылки на View, чтобы избежать утечек памяти.

`onDestroyView()` is called when the **Fragment's View is destroyed**, while the Fragment instance may still remain in memory.
This can happen when the Fragment is placed on the back stack or when its UI needs to be recreated.

This method is typically used to clear references to Views to avoid memory leaks.

```
override fun onDestroyView() {
    super.onDestroyView()
    // clear view references
}
```

🟡 Configuration changes

49. Что такое configuration change?

Что такое `configuration change`?
What is a `configuration change`?

`Configuration change` — это изменение параметров устройства, которое может повлиять на ресурсы приложения.
Например: поворот экрана, изменение языка, размера шрифта или подключение клавиатуры.

При configuration change Android обычно **пересоздаёт Activity**, чтобы загрузить ресурсы, соответствующие новой конфигурации.

A `configuration change` is a change in device settings that can affect the app's resources.
Examples include screen rotation, language change, font size change, or keyboard connection.

Ссылка на видео этого руководства на сайте: borisproit.expert

When a configuration change happens, Android usually **recreates the Activity** so that the correct resources for the new configuration can be loaded.

```
// Activity will be recreated during configuration change
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
}
```

50. Что происходит при повороте экрана?

Что происходит при повороте экрана?
What happens when the screen rotates?

При повороте экрана происходит **configuration change**, и Android по умолчанию **уничтожает и пересоздаёт Activity**.
Это нужно для загрузки ресурсов, подходящих для новой ориентации (например, `layout-land`).

Последовательность lifecycle обычно выглядит так:
`onPause()` → `onStop()` → `onDestroy()` → `onCreate()` → `onStart()` → `onResume()`.

When the screen rotates, a **configuration change** occurs and Android **destroys and recreates the Activity** by default.
This allows the system to load resources appropriate for the new orientation (for example `layout-land`).

The lifecycle sequence typically looks like this:
`onPause()` → `onStop()` → `onDestroy()` → `onCreate()` → `onStart()` → `onResume()`.

51. Почему Activity пересоздаётся при rotation?

Почему `Activity` пересоздаётся при `rotation`?
Why is an `Activity` recreated on screen rotation?

Activity пересоздаётся, потому что поворот экрана вызывает **configuration change**.
Android должен заново загрузить ресурсы, которые могут отличаться для новой ориентации (например, `layout`, `layout-land`, размеры, изображения).

Ссылка на видео этого руководства на сайте: borisproit.expert

Пересоздание Activity позволяет автоматически применить правильные ресурсы и перестроить UI.

The Activity is recreated because screen rotation triggers a **configuration change**.
Android needs to reload resources that may differ for the new orientation (for example `layout`, `layout-land`, dimensions, images).

Recreating the Activity allows the system to automatically apply the correct resources and rebuild the UI.

```
// Android loads different layout for landscape
res/layout/activity_main.xml
res/layout-land/activity_main.xml
```

52. Что происходит с ViewModel при rotation?

Что происходит с `ViewModel` при `rotation`?
What happens to a `ViewModel` during screen rotation?

`ViewModel` **не пересоздаётся** при rotation.
Когда Activity уничтожается из-за configuration change, система сохраняет ViewModel и передаёт её новой Activity.

Это позволяет сохранить данные и состояние UI при повороте экрана.

A `ViewModel` **is not recreated** during screen rotation.
When the Activity is destroyed due to a configuration change, the system retains the ViewModel and provides it to the new Activity instance.

This allows UI data and state to survive configuration changes.

53. Как сохранить состояние при configuration change?

Как сохранить состояние при `configuration change`?
How do you preserve state during a configuration change?

Состояние при `configuration change` обычно сохраняют тремя основными способами:

Ссылка на видео этого руководства на сайте: borisproit.expert

1. **ViewModel** — для данных UI, которые должны пережить поворот экрана.

2. **onSaveInstanceState()/Bundle`** — для небольших временных данных, которые нужно восстановить после пересоздания Activity или Fragment.

3. **SavedStateHandle** — если нужно сохранить данные внутри ViewModel и восстановить их после process death.

Обычно правило такое:
— UI state → `ViewModel`
— маленькие временные значения → `Bundle`
— state во ViewModel с восстановлением → `SavedStateHandle`

State during a configuration change is usually preserved in three main ways:

1. **ViewModel** — for UI data that should survive screen rotation.

2. **onSaveInstanceState()/Bundle`** — for small temporary values that must be restored after Activity or Fragment recreation.

3. **SavedStateHandle** — when state should be kept inside a ViewModel and restored even after process death.

A common rule is:
— UI state → `ViewModel`
— small temporary values → `Bundle`
— ViewModel state with restoration → `SavedStateHandle`

54. Что такое onSaveInstanceState()?

Что такое `onSaveInstanceState()`?
What is `onSaveInstanceState()`?

`onSaveInstanceState()` — это метод lifecycle Activity или Fragment, который используется для **сохранения временного состояния UI перед уничтожением компонента**.
Он вызывается перед тем, как Activity может быть уничтожена, например при `configuration change` (поворот экрана).

Ссылка на видео этого руководства на сайте: borisproit.expert

Данные сохраняются в `Bundle`, чтобы потом восстановить их в `onCreate()` или `onRestoreInstanceState()`.

`onSaveInstanceState()` is a lifecycle method of an Activity or Fragment used to **save temporary UI state before the component may be destroyed**.
It is called before the Activity might be destroyed, for example during a `configuration change` (screen rotation).

The state is stored in a `Bundle` and can later be restored in `onCreate()` or `onRestoreInstanceState()`.

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)

    outState.putString("username", "Alex") // save UI state
}

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    val name = savedInstanceState?.getString("username")
}
```

55. Когда вызывается onSaveInstanceState()?

Когда вызывается `onSaveInstanceState()`?
When is `onSaveInstanceState()` called?

`onSaveInstanceState()` вызывается, когда система **может уничтожить Activity**, но есть вероятность, что она будет восстановлена позже.
Например: при `configuration change` (поворот экрана) или когда приложение уходит в фон и система может освободить память.

Этот метод вызывается **перед `onStop()`**.

`onSaveInstanceState()` is called when the system **may destroy the Activity**, but there is a possibility it will be recreated later.

For example, during a `configuration change` (screen rotation) or when the app goes to the background and the system may reclaim memory.

This method is typically called **before `onStop()`**.

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)

    outState.putInt("counter", 10) // save state
}
```

56. Что такое SavedStateHandle?

Что такое `SavedStateHandle`?
What is `SavedStateHandle`?

`SavedStateHandle` — это компонент из Android Architecture Components, который позволяет **сохранять и восстанавливать состояние внутри ViewModel**.
Он работает как key-value хранилище и автоматически сохраняет данные в `Bundle`.

Главное отличие от обычной ViewModel — `SavedStateHandle` может восстановить данные **даже после process death**.

`SavedStateHandle` is a component from Android Architecture Components that allows you to **save and restore state inside a ViewModel**.
It works as a key-value storage and automatically persists data in a `Bundle`.

Unlike a regular ViewModel, `SavedStateHandle` can restore data **even after process death**.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
class UserViewModel(
    private val savedStateHandle: SavedStateHandle
) : ViewModel() {

    fun setQuery(query: String) {
        savedStateHandle["query"] = query // save value
    }

    fun getQuery(): String? {
        return savedStateHandle["query"]
    }
}
```

🟡 Fragment management

57. Что такое FragmentManager?

Что такое `FragmentManager`?
What is a `FragmentManager`?

`FragmentManager` — это компонент Android, который управляет жизненным циклом и транзакциями `Fragment` внутри `Activity`.
Он отвечает за добавление, удаление, замену и управление back stack фрагментов.

Через `FragmentManager` выполняются `FragmentTransaction`.

`FragmentManager` is an Android component that manages the lifecycle and transactions of `Fragments` inside an `Activity`.
It is responsible for adding, removing, replacing, and managing the fragment back stack.

`FragmentManager` is used to perform `FragmentTransaction`.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Replace Fragment using FragmentManager
supportFragmentManager.beginTransaction()
    .replace(R.id.container, UserFragment())
    .commit()
```

58. Что такое FragmentTransaction?

Что такое `FragmentTransaction`?
What is a `FragmentTransaction`?

`FragmentTransaction` — это объект, который используется для **выполнения операций над Fragment** через `FragmentManager`.
Он позволяет добавить, заменить, удалить или показать Fragment.

Транзакция собирает набор операций и применяется с помощью `commit()`.

A `FragmentTransaction` is an object used to **perform operations on Fragments** through the `FragmentManager`.
It allows adding, replacing, removing, or showing a Fragment.

The transaction collects a set of operations and applies them with `commit()`.

```
// Example FragmentTransaction
supportFragmentManager.beginTransaction()
    .replace(R.id.container, UserFragment()) // replace fragment
    .addToBackStack(null) // add to back stack
    .commit() // apply transaction
```

59. Чем replace отличается от add?

Чем `replace` отличается от `add`?
What is the difference between `replace` and `add`?

`add()` **добавляет новый Fragment**, не удаляя существующие.
В контейнере могут находиться несколько Fragment одновременно.

`replace()` **удаляет текущий Fragment из контейнера и добавляет новый**.
Фактически это комбинация операций `remove()` + `add()`.

Ссылка на видео этого руководства на сайте: borisproit.expert

---

`add()` **adds a new Fragment** without removing existing ones.
Multiple Fragments can exist in the same container.

`replace()` **removes the current Fragment from the container and adds a new one**.
It is effectively a combination of `remove()` + `add()`.

```
// add: existing fragment remains
supportFragmentManager.beginTransaction()
    .add(R.id.container, UserFragment())
    .commit()

// replace: existing fragment removed
supportFragmentManager.beginTransaction()
    .replace(R.id.container, UserFragment())
    .commit()
```

60. Что такое back stack?

Что такое `back stack`?
What is the `back stack`?

`Back stack` — это стек операций `FragmentTransaction`, который позволяет пользователю **возвращаться к предыдущему экрану при нажатии кнопки Back**.
Когда транзакция добавляется в back stack, предыдущий Fragment сохраняется и может быть восстановлен.

Если транзакция **не добавлена** в back stack, предыдущий Fragment будет удалён без возможности вернуться к нему.

The `back stack` is a stack of `FragmentTransaction` operations that allows the user to **navigate back to the previous screen when the Back button is pressed**.
When a transaction is added to the back stack, the previous Fragment is preserved and can be restored.

If a transaction is **not added** to the back stack, the previous Fragment is removed with no way to return to it.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
supportFragmentManager.beginTransaction()
    .replace(R.id.container, UserFragment())
    .addToBackStack(null) // add transaction to back stack
    .commit()
```

61. Что делает addToBackStack()?

Что делает `addToBackStack()`?
What does `addToBackStack()` do?

`addToBackStack()` добавляет текущую `FragmentTransaction` в **back stack**.
Это позволяет пользователю вернуться к предыдущему Fragment при нажатии кнопки Back.

Если не вызвать `addToBackStack()`, предыдущий Fragment будет удалён и назад вернуться нельзя.

`addToBackStack()` adds the current `FragmentTransaction` to the **back stack**.
This allows the user to return to the previous Fragment when the Back button is pressed.

If `addToBackStack()` is not used, the previous Fragment is removed and cannot be restored.

```
supportFragmentManager.beginTransaction()
    .replace(R.id.container, UserFragment())
    .addToBackStack(null) // allows navigating back
    .commit()
```

62. Чем popBackStack отличается от popBackStackImmediate()?

Чем `popBackStack()` отличается от `popBackStackImmediate()`?
What is the difference between `popBackStack()` and `popBackStackImmediate()`?

`popBackStack()` **запрашивает удаление последней транзакции из back stack**, но операция выполняется **асинхронно**.
Фактическое изменение происходит позже, когда FragmentManager обработает очередь операций.

`popBackStackImmediate()` делает то же самое, но **выполняет операцию сразу (синхронно)** и возвращает результат выполнения.

Ссылка на видео этого руководства на сайте: borisproit.expert

---

`popBackStack()` **requests the removal of the last transaction from the back stack**, but the operation is **performed asynchronously**.
The actual change happens later when the FragmentManager processes pending operations.

`popBackStackImmediate()` performs the same action but **executes it immediately (synchronously)** and returns whether the operation succeeded.

```
// Asynchronous
supportFragmentManager.popBackStack()

// Synchronous
supportFragmentManager.popBackStackImmediate()
```

🟡 Process death

63. Что такое process death?

Что такое `process death`?
What is `process death`?

`Process death` — это ситуация, когда система Android **полностью завершает процесс приложения**, чтобы освободить память или ресурсы.
После этого все объекты приложения уничтожаются, включая Activity, Fragment и ViewModel.

Когда пользователь возвращается в приложение, Android **создаёт новый процесс и восстанавливает состояние через saved state**.

`Process death` is a situation where the Android system **completely terminates the app process** to free memory or resources.
When this happens, all objects of the application are destroyed, including Activities, Fragments, and ViewModels.

When the user returns to the app, Android **creates a new process and restores state using saved state mechanisms**.

Ссылка на видео этого руководства на сайте: borisproit.expert

64. Когда Android может убить процесс приложения?

Когда Android может убить процесс приложения?
When can Android kill an app process?

Android может убить процесс приложения, когда системе **нужно освободить память или ресурсы**.
Это может произойти, даже если приложение не было закрыто пользователем.

Основные ситуации:

— **Нехватка памяти** — система завершает процессы в фоне, чтобы освободить RAM.
— **Приложение долго находится в фоне** — процесс может быть удалён из памяти.
— **Система управляет ресурсами устройства** — чтобы обеспечить стабильную работу других приложений.
— **Система завершает приложение после crash или ANR**.

Android can kill an app process when the system **needs to free memory or system resources**.
This can happen even if the user did not explicitly close the app.

Common situations include:

— **Low memory** — the system kills background processes to free RAM.
— **The app stays in the background for a long time** — its process may be removed from memory.
— **System resource management** — to keep other apps and the system stable.
— **After a crash or ANR**, the system may terminate the process.

65. Чем process death отличается от configuration change?

Чем `process death` отличается от `configuration change`?
What is the difference between `process death` and a `configuration change`?

`Configuration change` — это изменение параметров устройства (например, поворот экрана), при котором **Activity пересоздаётся**, но **процесс приложения остаётся живым**.
Большинство объектов в памяти сохраняются, например `ViewModel`.

`Process death` — это ситуация, когда **Android полностью уничтожает процесс приложения**.

Все объекты удаляются из памяти, и при следующем запуске приложение создаётся заново.

A `configuration change` is a change in device settings (such as screen rotation) where the **Activity is recreated**, but the **application process remains alive**.
Most in-memory objects remain available, including `ViewModel`.

`Process death` happens when **Android completely kills the app process**.
All objects are removed from memory, and the app must be recreated from scratch when opened again.

66. Как восстановить состояние после process death?

Как восстановить состояние после `process death`?
How do you restore state after `process death`?

После `process death` обычная `ViewModel` уже не поможет, потому что весь процесс приложения был уничтожен.
Для восстановления состояния используют:
— `onSaveInstanceState()` / `Bundle` — для небольших UI-данных
— `SavedStateHandle` — для состояния внутри `ViewModel`
— локальную базу данных / DataStore / SharedPreferences — для более важных или долгоживущих данных

Главная идея: всё, что нужно восстановить после полного уничтожения процесса, должно быть сохранено вне обычной памяти приложения.

After `process death`, a regular `ViewModel` is no longer enough because the entire app process has been destroyed.
To restore state, you typically use:
— `onSaveInstanceState()` / `Bundle` — for small UI values
— `SavedStateHandle` — for state inside a `ViewModel`
— local database / DataStore / SharedPreferences — for important or long-lived data

The key idea is that anything that must survive full process death has to be stored outside normal in-memory objects.

Ссылка на видео этого руководства на сайте: borisproit.expert

🟡 Context deeper

67. Когда нельзя хранить Activity Context?

Когда нельзя хранить `Activity Context`?
When should you not keep an `Activity Context`?

`Activity Context` нельзя хранить в **долгоживущих объектах**, потому что это может привести к **утечке памяти (memory leak)**.
Когда Activity уничтожается, ссылка на неё должна исчезнуть, иначе система не сможет освободить память.

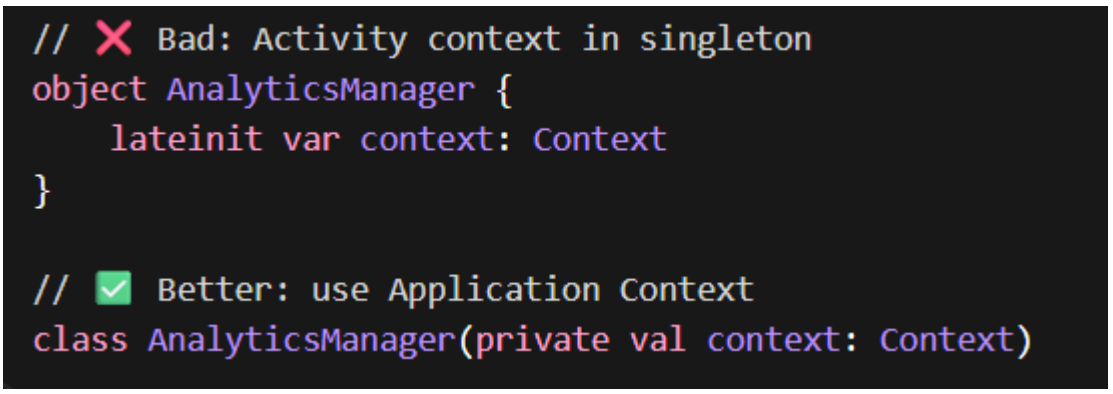
Нельзя хранить Activity Context в:
— `Singleton`
— `Repository`
— `ViewModel`
— статических (`static`) переменных

В таких случаях нужно использовать **Application Context**.

You should not keep an `Activity Context` in **long-living objects**, because it can cause **memory leaks**.
When an Activity is destroyed, references to it must be cleared so the system can free memory.

You should not store an Activity Context in:
— `Singleton`
— `Repository`
— `ViewModel`
— static variables

In these cases you should use the **Application Context** instead.



Ссылка на видео этого руководства на сайте: borisproit.expert

68. Почему Activity Context может вызвать memory leak?

Почему `Activity Context` может вызвать `memory leak`?
Why can an `Activity Context` cause a memory leak?

`Activity Context` содержит ссылку на всю Activity и её View.
Если долгоживущий объект (например Singleton или ViewModel) хранит ссылку на этот Context, Activity не сможет быть удалена из памяти после `onDestroy()`.

В результате Activity остаётся в памяти вместе со всем UI, что приводит к **memory leak**.

An `Activity Context` holds a reference to the entire Activity and its Views.
If a long-living object (such as a Singleton or ViewModel) keeps a reference to this Context, the Activity cannot be garbage collected after `onDestroy()`.

As a result, the Activity remains in memory along with its entire UI, causing a **memory leak**.

```
// ❌ Activity context stored in singleton
object Manager {
    var context: Context? = null
}

// If this is an Activity context,
// the Activity cannot be garbage collected
```

69. Когда нужно использовать Application Context?

Когда нужно использовать `Application Context`?
When should you use `Application Context`?

`Application Context` нужно использовать, когда объект **живет дольше Activity** и не зависит от UI.
Он существует столько же, сколько и всё приложение, поэтому безопасен для долгоживущих компонентов.

Типичные случаи использования:

— `Repository`
— `Singleton`
— `Database` (Room)
— `DI` контейнер
— доступ к системным сервисам

Ссылка на видео этого руководства на сайте: borisproit.expert

---

`Application Context` should be used when an object **lives longer than an Activity** and does not depend on the UI.
It lives as long as the entire application, making it safe for long-lived components.

Typical use cases include:

— `Repository`
— `Singleton`
— `Database` (Room)
— `DI` container
— accessing system services

```
// Using Application Context
val appContext = applicationContext

val manager = AnalyticsManager(appContext)
```

🟡 Storage

70. Что такое SQLite?

Что такое `SQLite`?
What is `SQLite`?

`SQLite` — это встроенная реляционная база данных, которая используется в Android для хранения данных на устройстве.
Она не требует отдельного сервера и хранит всю базу в одном файле внутри приложения.

SQLite позволяет выполнять SQL-запросы для создания таблиц, чтения, записи и обновления данных.

`SQLite` is a built-in relational database used in Android to store data locally on the device.
It does not require a separate server and stores the entire database in a single file within the application.

Ссылка на видео этого руководства на сайте: borisproit.expert

SQLite allows you to execute SQL queries to create tables, read, insert, update, and delete data.

71. Чем SQLite отличается от Room?

Чем `SQLite` отличается от `Room`?
What is the difference between `SQLite` and `Room`?

`SQLite` — это **низкоуровневая база данных**, где разработчик сам пишет SQL-запросы и управляет курсорами.
Работа напрямую с SQLite требует больше кода и легче допустить ошибки.

`Room` — это **библиотека из Android Jetpack**, которая является abstraction layer над SQLite.
Она предоставляет типобезопасные DAO, аннотации и автоматическую генерацию кода для работы с базой данных.

`SQLite` is a **low-level database** where developers manually write SQL queries and work with cursors.
Direct SQLite usage requires more boilerplate and is more error-prone.

`Room` is a **Jetpack library** that acts as an abstraction layer on top of SQLite.
It provides type-safe DAOs, annotations, and automatic code generation for database operations.

72. Когда использовать Room вместо SQLite?

Когда использовать `Room` вместо `SQLite`?
When should you use `Room` instead of `SQLite`?

`Room` используют в большинстве Android-приложений, потому что он упрощает работу с базой данных и уменьшает количество ошибок.
Он предоставляет type-safe API, DAO, аннотации и автоматическую генерацию кода.

`SQLite` напрямую используют редко — обычно только когда нужен полный контроль над SQL или очень специфическая оптимизация.

Ссылка на видео этого руководства на сайте: borisproit.expert

`Room` is used in most Android applications because it simplifies database work and reduces errors.
It provides a type-safe API, DAOs, annotations, and automatic code generation.

Direct `SQLite` is rarely used today and is typically chosen only when full control over SQL or very specific optimizations are required.

73. Какие преимущества дает Room?

Какие преимущества даёт `Room`?
What advantages does `Room` provide?

`Room` упрощает работу с базой данных и делает код более безопасным и удобным для поддержки.

Основные преимущества:

— **Type safety** — SQL-запросы проверяются на этапе компиляции.
— **Меньше boilerplate-кода** — не нужно вручную работать с `Cursor`.
— **DAO интерфейсы** — удобный и структурированный доступ к данным.
— **Интеграция с Coroutines и Flow** — можно получать данные асинхронно.
— **Поддержка LiveData / Flow** — автоматические обновления UI при изменении данных.

`Room` simplifies working with a database and makes the code safer and easier to maintain.

Main advantages:

— **Type safety** — SQL queries are validated at compile time.
— **Less boilerplate** — no need to manually work with `Cursor`.
— **DAO interfaces** — structured and convenient data access.
— **Integration with Coroutines and Flow** — asynchronous data access.
— **LiveData / Flow support** — automatic UI updates when data changes.

🟡 Background work

74. Что такое WorkManager?

Ссылка на видео этого руководства на сайте: borisproit.expert

Что такое `WorkManager`?
What is `WorkManager`?

`WorkManager` — это библиотека из Android Jetpack для выполнения **отложенных или фоновых задач**, которые должны быть гарантированно выполнены.
Он используется для задач, которые должны выполниться даже если приложение закрыто или устройство перезапущено.

WorkManager автоматически выбирает лучший механизм выполнения (JobScheduler, AlarmManager или Foreground Service).

`WorkManager` is a Jetpack library used to run **deferrable background tasks** that are guaranteed to execute.
It is designed for work that should run even if the app is closed or the device restarts.

WorkManager automatically chooses the best execution mechanism (JobScheduler, AlarmManager, or Foreground Service).

75. Чем WorkManager отличается от Service?

Чем `WorkManager` отличается от `Service`?
What is the difference between `WorkManager` and a `Service`?

`Service` — это компонент Android, который выполняет задачи в фоне **пока приложение активно или запущен сервис**.
Он используется для долгих операций, связанных с текущей работой приложения (например, музыка или навигация).

`WorkManager` — это библиотека для **гарантированного выполнения фоновых задач**, даже если приложение закрыто или устройство перезапущено.

A `Service` is an Android component used to perform background work **while the app or service is running**.
It is typically used for long-running tasks related to the current user session (for example music playback or navigation).

`WorkManager` is a library used for **guaranteed background work**, even if the app is closed or the device restarts.

Ссылка на видео этого руководства на сайте: borisproit.expert

76. Чем WorkManager отличается от Executor?

Чем `WorkManager` отличается от `Executor`?
What is the difference between `WorkManager` and an `Executor`?

`Executor` — это инструмент Java/Kotlin для **выполнения задач в другом потоке внутри текущего процесса приложения**.
Он работает только пока приложение запущено.

`WorkManager` — это система Android для **планирования и гарантированного выполнения фоновых задач**, даже если приложение закрыто или процесс был убит.

An `Executor` is a Java/Kotlin tool used to **run tasks on another thread inside the current app process**.
It only works while the application process is alive.

`WorkManager` is an Android system component used to **schedule and guarantee background work**, even if the app is closed or the process is killed.

77. Что такое Job Scheduling в Android?

Что такое `Job Scheduling` в Android?
What is `Job Scheduling` in Android?

`Job Scheduling` — это механизм Android для **планирования и выполнения фоновых задач в подходящее время** с учётом состояния устройства.
Система может отложить выполнение задачи, чтобы экономить батарею и ресурсы.

Например, задача может быть выполнена только когда устройство подключено к Wi-Fi или находится на зарядке.

`Job Scheduling` in Android is a mechanism for **scheduling and executing background tasks at an appropriate time**, based on device conditions.
The system may delay tasks to optimize battery usage and system resources.

For example, a job can be scheduled to run only when the device is connected to Wi-Fi or charging.

Ссылка на видео этого руководства на сайте: borisproit.expert

🟡 Jetpack

81. Что такое Android Jetpack?

Что такое Android Jetpack?
What is Android Jetpack?

Android Jetpack — это набор библиотек и инструментов от Google, который помогает разработчикам создавать Android-приложения быстрее, безопаснее и с меньшим количеством boilerplate-кода.
Jetpack предоставляет готовые решения для архитектуры, UI, навигации, фоновых задач и управления жизненным циклом.

Android Jetpack is a suite of libraries and tools from Google designed to help developers build Android apps faster, safer, and with less boilerplate code.
Jetpack provides ready-to-use components for architecture, UI, navigation, background work, and lifecycle management.

Основные категории Jetpack:

— **Architecture** — ViewModel, LiveData, Room, WorkManager
— **UI** — Jetpack Compose, Navigation, Fragment
— **Foundation** — AppCompat, RecyclerView
— **Behavior** — Permissions, Sharing

82. Какие основные библиотеки Jetpack ты знаешь?

Какие основные библиотеки Jetpack ты знаешь?
What are the main Jetpack libraries you know?

Основные библиотеки Android Jetpack можно разделить на несколько категорий.

Architecture:
— `ViewModel` — хранение и управление UI state
— `LiveData` — observable данные
— `Room` — работа с базой данных
— `WorkManager` — выполнение фоновых задач
— `DataStore` — хранение настроек
— `Paging` — загрузка данных страницами

UI:
— `Jetpack Compose` — декларативный UI

Ссылка на видео этого руководства на сайте: borisproit.expert

— `Navigation` — навигация между экранами
— `Fragment` — модульный UI
— `RecyclerView` — список элементов

Foundation:
— `AppCompat` — совместимость с разными версиями Android
— `Core KTX` — Kotlin-расширения для Android API

Behavior:
— `Lifecycle` — управление жизненным циклом компонентов
— `Hilt` — dependency injection
— `CameraX` — работа с камерой

The main Android Jetpack libraries can be grouped into several categories.

Architecture:
— `ViewModel` — stores and manages UI state
— `LiveData` — observable data holder
— `Room` — database abstraction
— `WorkManager` — background task scheduling
— `DataStore` — preferences/data storage
— `Paging` — load data in pages

UI:
— `Jetpack Compose` — declarative UI toolkit
— `Navigation` — screen navigation
— `Fragment` — modular UI components
— `RecyclerView` — list rendering

Foundation:
— `AppCompat` — backward compatibility
— `Core KTX` — Kotlin extensions for Android APIs

Behavior:
— `Lifecycle` — lifecycle-aware components
— `Hilt` — dependency injection
— `CameraX` — camera APIs

Ссылка на видео этого руководства на сайте: borisproit.expert

🟡 Security

83. Как можно защитить данные приложения?

Как можно защитить данные приложения?
How can application data be protected?

Данные приложения защищают на нескольких уровнях:
— хранить чувствительные данные в **EncryptedSharedPreferences** или **EncryptedFile**
— использовать **Room/SQLite** вместе с шифрованием, если хранятся важные локальные данные
— не хранить токены, пароли и секреты в открытом виде
— использовать **Android Keystore** для хранения ключей шифрования
— передавать данные по сети только через **HTTPS**
— ограничивать доступ к `ContentProvider`, файлам и экспортируемым компонентам
— использовать `internal storage`, если данные не должны быть доступны другим приложениям

Главная идея: **чувствительные данные нужно шифровать и минимизировать доступ к ним**.

Application data should be protected on multiple levels:
— store sensitive data in **EncryptedSharedPreferences** or **EncryptedFile**
— use **Room/SQLite** with encryption if important local data is stored
— never store tokens, passwords, or secrets in plain text
— use **Android Keystore** to store encryption keys
— send data over the network only via **HTTPS**
— restrict access to `ContentProvider`, files, and exported components
— use `internal storage` when data should not be accessible to other apps

The main idea is: **sensitive data should be encrypted and access should be minimized**.

84. Что такое Android Keystore?

Что такое `Android Keystore`?
What is the `Android Keystore`?

`Android Keystore` — это защищённое хранилище системы Android, предназначенное для **безопасного хранения криптографических ключей**.
Ключи хранятся внутри системы и не могут быть извлечены приложением в открытом виде.

Ссылка на видео этого руководства на сайте: borisproit.expert

Это позволяет выполнять операции шифрования, расшифрования и подписи данных, не раскрывая сам ключ.

`Android Keystore` is a secure storage system in Android used to **store cryptographic keys safely**.
The keys are kept inside the system and cannot be extracted by the application in plain form.

This allows apps to perform encryption, decryption, and signing operations without exposing the key itself.

85. Как хранить API-ключи безопасно?

Как хранить API-ключи безопасно?
How should API keys be stored securely?

API-ключи нельзя считать полностью безопасными в клиентском Android-приложении, потому что APK можно декомпилировать.
Поэтому главный принцип такой: **секретные ключи не должны храниться на клиенте**, если от них зависит безопасность системы.

Что делать правильно:
— хранить секретные ключи на **backend server**
— Android-приложению отдавать только временный токен или обращаться к своему backend
— для локально используемых ключей не хардкодить их прямо в код
— не хранить ключи в Git-репозитории
— использовать `local.properties`, `BuildConfig`, CI/CD secrets только как базовую защиту, но не как настоящую безопасность
— если ключ всё же попадает на клиент, ограничивать его по domain / IP / package name / SHA certificate fingerprint, если сервис это поддерживает
— для чувствительных данных использовать `Android Keystore` или шифрование, но понимать, что это не спасает секрет, который должен работать на клиенте

API keys can never be considered fully secure in a client-side Android app because the APK can be reverse engineered.
So the main rule is: **do not store truly secret keys on the client** if system security depends on them.

What to do instead:
— keep secret keys on a **backend server**
— let the Android app call your backend or receive temporary tokens

— avoid hardcoding keys directly in source code
— never commit keys to Git
— use `local.properties`, `BuildConfig`, and CI/CD secrets only as basic hygiene, not as real protection
— if a key must be present on the client, restrict it by domain / IP / package name / SHA certificate fingerprint when the provider supports it
— use `Android Keystore` or encryption for sensitive local data, but understand this does not fully protect a secret that must run on the client

86. Что такое ProGuard и зачем он используется?

Что такое `ProGuard` и зачем он используется?
What is `ProGuard` and why is it used?

`ProGuard` — это инструмент, который используется при сборке Android-приложения для **оптимизации, обфускации и уменьшения размера кода**.
Он удаляет неиспользуемый код, сокращает имена классов и методов и усложняет обратную разработку APK.

Основные цели ProGuard:

— **Минификация (code shrinking)** — удаление неиспользуемого кода
— **Обфускация (obfuscation)** — изменение имён классов, методов и полей
— **Оптимизация** — упрощение и оптимизация байткода
— **Уменьшение размера APK**

`ProGuard` is a tool used during the Android build process for **code shrinking, obfuscation, and optimization**.
It removes unused code, renames classes and methods, and makes reverse engineering harder.

Main purposes of ProGuard:

— **Code shrinking** — removes unused classes and methods
— **Obfuscation** — renames classes, methods, and fields
— **Optimization** — simplifies and optimizes bytecode
— **Reduces APK size**

Ссылка на видео этого руководства на сайте: borisproit.expert

🟡 UI system

87. Что такое View в Android?

Что такое `View` в Android?
What is a `View` in Android?

`View` — это базовый класс для всех элементов пользовательского интерфейса в Android.
Он представляет **прямоугольную область на экране**, которая отвечает за отображение и обработку пользовательских действий.

К примеру, кнопки, текстовые поля и изображения — всё это наследники класса `View`.

A `View` is the base class for all UI elements in Android.
It represents a **rectangular area on the screen** responsible for drawing content and handling user interactions.

Examples such as buttons, text fields, and images are all subclasses of `View`.

```
// Example View
val button = findViewById<Button>(R.id.myButton)

button.setOnClickListener {
    // handle click
}
```

88. Что такое ViewGroup?

Что такое `ViewGroup`?
What is a `ViewGroup`?

`ViewGroup` — это специальный тип `View`, который используется как **контейнер для других View**.
Он управляет расположением (layout) и иерархией дочерних элементов интерфейса.

Все layout-компоненты Android (например `LinearLayout`, `FrameLayout`, `ConstraintLayout`) наследуются от `ViewGroup`.

Ссылка на видео этого руководства на сайте: borisproit.expert

A `ViewGroup` is a special type of `View` that acts as a **container for other Views**.
It manages the layout and hierarchy of its child UI elements.

All Android layout components (such as `LinearLayout`, `FrameLayout`, `ConstraintLayout`) extend `ViewGroup`.

89. Что такое View hierarchy?

Что такое View hierarchy?
What is the View hierarchy?

View hierarchy — это древовидная структура UI, состоящая из `View` и `ViewGroup`.
`ViewGroup` выступает контейнером и может содержать другие `View` или `ViewGroup`, формируя дерево интерфейса.

Корнем этой структуры обычно является layout Activity или Fragment.

A **View hierarchy** is a tree structure of UI elements composed of `View` and `ViewGroup`.
A `ViewGroup` acts as a container and can hold other `View` or `ViewGroup` elements, forming a hierarchical tree.

The root of this hierarchy is typically the layout of an Activity or Fragment.

```
<!-- Example View hierarchy -->
LinearLayout
    TextView
    Button
    ImageView
```

90. Почему глубокая View hierarchy ухудшает performance?

Почему глубокая View hierarchy ухудшает `performance`?
Why does a deep View hierarchy hurt performance?

Глубокая View hierarchy ухудшает производительность, потому что Android должен **обходить всё дерево View при каждом layout и draw проходе**.
Чем больше уровней вложенности, тем больше вычислений требуется системе.

Каждый View проходит три основных этапа:
— **measure** — вычисление размеров

Ссылка на видео этого руководства на сайте: borisproit.expert

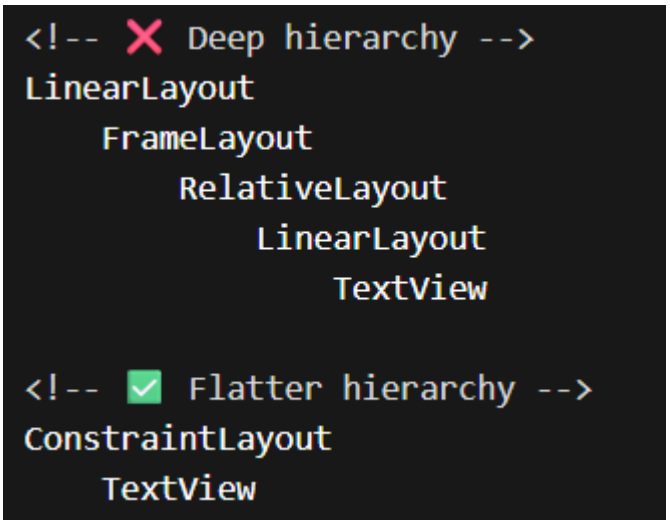
— **layout** — позиционирование
— **draw** — отрисовка

Глубокая иерархия увеличивает количество этих операций и может замедлять UI.

A deep View hierarchy hurts performance because Android must **traverse the entire view tree during layout and drawing passes**.
The more nested levels there are, the more work the system has to do.

Each View goes through three main phases:
— **measure** — calculating size
— **layout** — positioning
— **draw** — rendering

A deep hierarchy increases the number of these operations and can slow down the UI.



91. Как оптимизировать layout hierarchy?

Как оптимизировать `layout hierarchy`?
How can you optimize the `layout hierarchy`?

Иерархию layout оптимизируют за счёт **уменьшения количества вложенных ViewGroup**.
Чем структура UI проще и “площе”, тем меньше работы Android выполняет на этапах `measure`, `layout` и `draw`.

Основные способы оптимизации:
— использовать **ConstraintLayout** вместо нескольких вложенных `LinearLayout` / `RelativeLayout`
— удалять лишние контейнеры
— использовать `<merge>` там, где не нужен дополнительный корневой layout

Ссылка на видео этого руководства на сайте: borisproit.expert

— переиспользовать части UI через `<include>`
— избегать слишком глубокой вложенности
— для списков использовать `RecyclerView`, а не большое количество View вручную

You optimize a layout hierarchy by **reducing the number of nested ViewGroups**.
The simpler and flatter the UI structure is, the less work Android has to do during `measure`, `layout`, and `draw`.

Main optimization techniques:
— use **ConstraintLayout** instead of multiple nested `LinearLayout` / `RelativeLayout`
— remove unnecessary containers
— use `<merge>` when an extra root layout is not needed
— reuse UI parts with `<include>`
— avoid deep nesting
— use `RecyclerView` for lists instead of manually adding many Views

```
<!-- ❌ Deep hierarchy -->
<LinearLayout>
    <LinearLayout>
        <TextView />
        <Button />
    </LinearLayout>
</LinearLayout>

<!-- ✅ Flatter hierarchy -->
<androidx.constraintlayout.widget.ConstraintLayout>
    <TextView />
    <Button />
</androidx.constraintlayout.widget.ConstraintLayout>
```

🟡 RecyclerView

92. Чем ListView отличается от RecyclerView?

Чем `ListView` отличается от `RecyclerView`?
What is the difference between `ListView` and `RecyclerView`?

Ссылка на видео этого руководства на сайте: borisproit.expert

`ListView` — это старый компонент Android для отображения списка элементов.
Он имеет ограниченные возможности и менее гибкую архитектуру.

`RecyclerView` — более современный и гибкий компонент, который использует **ViewHolder pattern** и позволяет эффективно переиспользовать элементы списка.

`ListView` is an older Android component used to display lists of items.
It has limited capabilities and a less flexible architecture.

`RecyclerView` is a newer and more flexible component that uses the **ViewHolder pattern** and efficiently recycles list item views.

Основные отличия:

— `RecyclerView` требует `Adapter` и `ViewHolder`, а в `ListView` ViewHolder был опциональным
— `RecyclerView` поддерживает разные `LayoutManager` (vertical, grid, staggered)
— `RecyclerView` поддерживает анимации и декорации элементов
— `RecyclerView` лучше оптимизирован для больших списков

Key differences:

— `RecyclerView` requires an `Adapter` and `ViewHolder`, while `ListView` used ViewHolder optionally
— `RecyclerView` supports different `LayoutManager` types (vertical, grid, staggered)
— `RecyclerView` supports item animations and decorations
— `RecyclerView` is better optimized for large lists

93. Как работает RecyclerView?

Как работает `RecyclerView`?
How does `RecyclerView` work?

`RecyclerView` работает по принципу **переиспользования View (recycling)**.
Вместо создания View для каждого элемента списка, он создаёт только несколько View и **переиспользует их при прокрутке**.

Когда элемент выходит за пределы экрана, его View не уничтожается — оно возвращается в **recycle pool** и используется для нового элемента.

Основные компоненты:

Ссылка на видео этого руководства на сайте: borisproit.expert

— **Adapter** — предоставляет данные
— **ViewHolder** — хранит ссылки на View элементов
— **LayoutManager** — определяет расположение элементов на экране
— **Recycler** — переиспользует View

`RecyclerView` works using the **view recycling mechanism**.
Instead of creating a View for every list item, it creates only a limited number of Views and **reuses them while scrolling**.

When an item scrolls off the screen, its View is not destroyed — it goes to a **recycle pool** and is reused for another item.

Main components:

— **Adapter** — provides data
— **ViewHolder** — holds references to item Views
— **LayoutManager** — controls how items are positioned
— **Recycler** — reuses existing Views

94. Что такое ViewHolder pattern?

Что такое ViewHolder pattern?
What is the ViewHolder pattern?

`ViewHolder pattern` — это паттерн оптимизации UI, при котором **ссылки на View элементов списка сохраняются в специальном объекте**, чтобы не выполнять `findViewById()` каждый раз при обновлении элемента.

Это уменьшает количество дорогостоящих операций поиска View и улучшает производительность списков.

The **ViewHolder pattern** is a UI optimization pattern where **references to item Views are stored inside a dedicated object**, so `findViewById()` does not need to be called repeatedly when binding data.

This reduces expensive view lookups and improves list performance.

🟡 Navigation

Ссылка на видео этого руководства на сайте: borisproit.expert

97. Что такое deep link?

Что такое deep link?
What is a deep link?

Deep link — это ссылка, которая открывает **конкретный экран внутри приложения**, а не просто запускает приложение.
Она позволяет пользователю сразу перейти к нужному контенту.

Deep links часто используются из браузера, email, push-уведомлений или других приложений.

A **deep link** is a link that opens a **specific screen inside an app**, instead of just launching the app.
It allows users to navigate directly to particular content.

Deep links are commonly used from browsers, emails, push notifications, or other apps.

```
<!-- Deep link example in AndroidManifest -->
<intent-filter>
    <action android:name="android.intent.action.VIEW"/>

    <category android:name="android.intent.category.DEFAULT"/>
    <category android:name="android.intent.category.BROWSABLE"/>

    <data android:scheme="https"
          android:host="example.com"
          android:pathPrefix="/profile"/>
</intent-filter>
```

98. Чем deep link отличается от app link?

Чем deep link отличается от app link?
What is the difference between a deep link and an app link?

Deep link — это любая ссылка, которая открывает конкретный экран приложения.
Она может использовать **кастомную схему** (`myapp://`) или обычный URL.

Проблема deep link в том, что система может предложить пользователю **выбрать приложение**, которое откроет ссылку.

Ссылка на видео этого руководства на сайте: borisproit.expert

App link — это специальный тип deep link, который использует **HTTPS URL и проверку домена**.
Если домен подтверждён (`assetlinks.json`), Android автоматически открывает ссылку **в приложении без выбора**.

An **app link** is a special type of deep link that uses **HTTPS URLs with domain verification**.
If the domain is verified (`assetlinks.json`), Android opens the link **directly in the app without asking the user**.

A **deep link** is any link that opens a specific screen inside an app.
It can use a **custom scheme** (`myapp://`) or a regular URL.

The downside is that Android may ask the user **which app should handle the link**.

```
<!-- App Link example -->
<intent-filter android:autoVerify="true">
    <action android:name="android.intent.action.VIEW"/>

    <category android:name="android.intent.category.DEFAULT"/>
    <category android:name="android.intent.category.BROWSABLE"/>

    <data
        android:scheme="https"
        android:host="example.com"/>
</intent-filter>
```

🟡 Debugging

99. Что такое Logcat?

Что такое `Logcat`?
What is `Logcat`?

`Logcat` — это система логирования в Android, которая выводит сообщения от приложения, системы и библиотек.
Она используется разработчиками для **отладки, анализа ошибок и отслеживания работы приложения**.

Ссылка на видео этого руководства на сайте: borisproit.expert

Через Logcat можно видеть ошибки, предупреждения и информационные сообщения во время работы приложения.

`Logcat` is the Android logging system that displays messages from the app, the system, and libraries.
It is used by developers for **debugging, error analysis, and monitoring app behavior**.

Through Logcat you can see errors, warnings, and informational messages while the app is running.

```
// Log message example
Log.d("MainActivity", "User clicked button")
```

🔴 Performance / Memory

100. Что такое memory leak в Android?

Что такое memory leak в Android?
What is a memory leak in Android?

Memory leak — это ситуация, когда объект больше не нужен приложению, но **ссылка на него всё ещё сохраняется**, поэтому сборщик мусора (Garbage Collector) не может освободить память.

В результате память постепенно заполняется, что может привести к **замедлению приложения или crash из-за OutOfMemoryError**.

A **memory leak** occurs when an object is no longer needed but **a reference to it is still kept**, preventing the Garbage Collector from freeing the memory.

As a result, memory usage grows over time and may lead to **performance issues or an OutOfMemoryError crash**.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// ❌ Memory leak example
object Manager {
    var activity: Activity? = null
}

// If an Activity is stored here,
// it cannot be garbage collected
```

101. Как можно создать memory leak в Android?

Как можно создать memory leak в Android?
How can a memory leak occur in Android?

`Memory leak` возникает, когда **долгоживущий объект хранит ссылку на короткоживущий объект**, например `Activity` или `View`.
В таком случае Garbage Collector не может освободить память после уничтожения Activity.

Типичные причины memory leak:

— хранение `Activity Context` в `Singleton`
— хранение `View` или `Activity` в `static` переменной
— `Handler`, `Runnable` или `Thread`, которые продолжают работать после уничтожения Activity
— `Listener`, который не был удалён
— долгоживущие `Coroutine` или `Flow`, привязанные к Activity

A `memory leak` occurs when a **long-living object holds a reference to a short-living object**, such as an `Activity` or `View`.
In this case, the Garbage Collector cannot free the memory after the Activity is destroyed.

Common causes of memory leaks:

— storing an `Activity Context` in a `Singleton`
— keeping `View` or `Activity` in a static variable
— `Handler`, `Runnable`, or `Thread` that continues running after the Activity is destroyed
— a `Listener` that was not removed
— long-running `Coroutine` or `Flow` tied to an Activity lifecycle

Ссылка на видео этого руководства на сайте: borisproit.expert

102. Как избежать memory leak?

Как избежать memory leak?
How can you avoid memory leaks?

Чтобы избежать memory leak, нужно следить за тем, чтобы **долгоживущие объекты не держали ссылки на короткоживущие**, такие как `Activity`, `Fragment` или `View`.

Основные правила:

— не хранить `Activity Context` в `Singleton` или `ViewModel`
— использовать **Application Context**, если объект живёт долго
— очищать `listeners`, `callbacks` и `observers`
— отменять `coroutines`, `threads` и `handlers` при уничтожении компонента
— использовать **lifecycle-aware компоненты** (`viewModelScope`, `lifecycleScope`)
— не хранить `View` в статических переменных

To avoid memory leaks, ensure that **long-living objects do not keep references to short-lived components** such as `Activity`, `Fragment`, or `View`.

Main practices:

— do not store `Activity Context` in `Singleton` or `ViewModel`
— use **Application Context** for long-lived objects
— clear `listeners`, `callbacks`, and `observers`
— cancel `coroutines`, `threads`, and `handlers` when the component is destroyed
— use **lifecycle-aware components** (`viewModelScope`, `lifecycleScope`)
— avoid storing `View` in static variables

103. Что такое Garbage Collector и как он работает **в** Android?

Что такое Garbage Collector и как он работает в Android?
What is the Garbage Collector and how does it work in Android?

Garbage Collector (GC) — это механизм управления памятью, который автоматически **освобождает память от объектов, на которые больше нет ссылок**.
Разработчику не нужно вручную удалять объекты — система делает это автоматически.

GC периодически анализирует объекты в памяти и проверяет, есть ли на них ссылки из **root объектов** (например, стека потоков или статических переменных).
Если на объект нет доступных ссылок, он считается **недостижимым (unreachable)** и память освобождается.

Ссылка на видео этого руководства на сайте: borisproit.expert

---

The **Garbage Collector (GC)** is a memory management mechanism that automatically **frees memory from objects that are no longer referenced**.
Developers do not need to manually delete objects — the system handles it automatically.

GC periodically scans objects in memory and checks whether they are reachable from **root objects** (such as thread stacks or static references).
If an object has no reachable references, it is considered **unreachable** and its memory is reclaimed.

🔴 System / OS

105. Что такое ANR?

Что такое ANR?
What is ANR?

ANR (Application Not Responding) — это ситуация, когда приложение **не отвечает на действия пользователя** в течение определённого времени.
Android считает приложение зависшим и показывает системное окно **“App isn't responding”**.

Чаще всего это происходит, когда **главный поток (Main/UI thread)** блокируется долгой операцией.

ANR (Application Not Responding) occurs when an app **does not respond to user input** for a certain period of time.
Android considers the app frozen and shows the system dialog **“App isn't responding”**.

This usually happens when the **main thread (UI thread)** is blocked by a long-running operation.

Типичные причины:

— сетевые запросы на `Main thread`
— работа с базой данных на `Main thread`
— сложные вычисления в UI потоке
— бесконечные циклы
— долгие операции ввода-вывода

Ссылка на видео этого руководства на сайте: borisproit.expert

---

Common causes:

— network requests on the `Main thread`
— database operations on the `Main thread`
— heavy computations on the UI thread
— infinite loops
— long I/O operations

106. Как можно диагностировать ANR?

Как можно диагностировать `ANR`?
How can you diagnose an `ANR`?

ANR диагностируют через логи, traces и анализ того, **что блокировало Main thread**. Главная цель — найти, почему UI-поток слишком долго не мог обработать ввод пользователя или системное событие.

Основные способы диагностики:

— смотреть **Logcat** — там могут быть сообщения об ANR
— анализировать файл **traces.txt / ANR traces** — он показывает, чем были заняты потоки в момент зависания
— проверять, не выполнялись ли на `Main thread`:
— сетевые запросы
— запросы к базе данных
— тяжёлые вычисления
— `Thread.sleep()`
— длинные циклы
— использовать **Android Studio Profiler** для анализа CPU и Main thread
— смотреть **ANR reports** в Google Play Console / Firebase Crashlytics, если приложение уже выпущено

ANRs are diagnosed through logs, traces, and by analyzing **what blocked the Main thread**. The main goal is to find why the UI thread could not process user input or a system event for too long.

Common ways to diagnose ANRs:

— check **Logcat** for ANR-related messages
— inspect **traces.txt / ANR traces** to see what threads were doing at the time of the freeze
— verify that the `Main thread` was not doing:

— network calls
— database work
— heavy computations
— `Thread.sleep()`
— long loops
— use **Android Studio Profiler** to inspect CPU usage and Main thread activity
— review **ANR reports** in Google Play Console / Firebase Crashlytics for production apps

---

🔴 Tasks / navigation system

107. Что такое task и back stack?

Что такое `task` и `back stack`?
What are a `task` and the `back stack`?

Task — это стек `Activity`, который представляет пользовательскую сессию выполнения приложения.
Он содержит набор Activity, между которыми пользователь перемещается.

Back stack — это порядок Activity внутри task, который определяет, **куда вернётся пользователь при нажатии кнопки Back**.

Когда новая Activity запускается, она помещается **на вершину back stack**.
При нажатии Back текущая Activity уничтожается, и система возвращается к предыдущей.

A **task** is a stack of `Activity` instances that represents a user session in an application.
It contains the set of Activities the user interacts with.

The **back stack** is the order of Activities inside a task that determines **which screen the user returns to when pressing the Back button**.

When a new Activity is launched, it is placed **on top of the back stack**.
When the Back button is pressed, the current Activity is destroyed and the system returns to the previous one.

108. Что такое launch modes?

Ссылка на видео этого руководства на сайте: borisproit.expert

Что такое launch modes?
What are launch modes?

Launch mode — это настройка в Android, которая определяет, **как Activity создаётся и добавляется в back stack** при запуске.

Она управляет тем, создаётся ли новая Activity или используется уже существующая.

A **launch mode** is an Android configuration that defines **how an Activity is created and placed in the back stack** when it is launched.

It controls whether a new instance is created or an existing one is reused.

Основные launch modes:

— **standard** — создаётся новая Activity каждый раз
— **singleTop** — если Activity уже находится на вершине back stack, новая не создаётся
— **singleTask** — в task может существовать только один экземпляр Activity
— **singleInstance** — Activity запускается в отдельном task

Main launch modes:

— **standard** — a new Activity instance is created every time
— **singleTop** — if the Activity is already at the top of the stack, a new instance is not created
— **singleTask** — only one instance of the Activity exists in the task
— **singleInstance** — the Activity runs in its own separate task

```
<activity
    android:name=".MainActivity"
    android:launchMode="singleTop" />
```

109. Чем standard отличается от singleTop?

Чем `standard` отличается от `singleTop`?
What is the difference between `standard` and `singleTop`?

`standard` — это режим запуска по умолчанию.
Каждый раз при запуске Activity создаётся **новый экземпляр**, даже если такая Activity уже находится на вершине back stack.

`singleTop` — новый экземпляр **не создаётся**, если Activity уже находится **на вершине back stack**.
В этом случае система вызывает метод `onNewIntent()` у существующей Activity.

`standard` is the default launch mode.
Every time the Activity is started, **a new instance is created**, even if the same Activity is already at the top of the back stack.

`singleTop` does **not create a new instance** if the Activity is already **on top of the back stack**.
Instead, the existing Activity receives the new intent through `onNewIntent()`.

110. Чем singleTask отличается от singleInstance?

Чем `singleTask` отличается от `singleInstance`?

`singleTask` — Activity существует **в одном экземпляре внутри task**.
Если такая Activity уже существует, Android **использует существующий экземпляр** и удаляет все Activity, которые находятся выше неё в back stack.

При этом в том же task **могут находиться другие Activity**.

`singleInstance` — Activity всегда запускается **в отдельном task**, и **никакие другие Activity не могут находиться в этом task**.
Она полностью изолирована от других Activity.

What is the difference between `singleTask` and `singleInstance`?

`singleTask` means the Activity exists **as a single instance within a task**.
If that Activity already exists, Android **reuses the existing instance** and removes all Activities that are above it in the back stack.

Other Activities **can still exist in the same task**.

`singleInstance` means the Activity always runs **in its own separate task**, and **no other Activities can exist in that task**.
It is completely isolated from other Activities.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
<activity
    android:name=".MainActivity"
    android:launchMode="singleTask"/>
```

🔴 Android system interaction

112. Что такое Binder?

Что такое `Binder`?

`Binder` — это механизм **межпроцессного взаимодействия (IPC)** в Android, который позволяет разным приложениям или компонентам системы **вызывать методы друг друга, как будто они находятся в одном процессе**.

Android использует Binder для общения между процессами, например между приложением и системными сервисами (`ActivityManager`, `LocationManager`, `NotificationManager`).

Когда приложение вызывает системный сервис, запрос проходит через **Binder driver в Linux Kernel**, который передаёт данные между процессами.

What is `Binder`?

`Binder` is the **inter-process communication (IPC)** mechanism used in Android that allows different processes to **call methods on each other as if they were in the same process**.

Android uses Binder for communication between apps and system services such as `ActivityManager`, `LocationManager`, and `NotificationManager`.

When an app calls a system service, the request goes through the **Binder driver in the Linux Kernel**, which transfers data between processes.

```
// Getting a system service through Binder IPC
val locationManager = getSystemService(Context.LOCATION_SERVICE) as LocationManager
```

113. Как приложения взаимодействуют друг с другом?

Как приложения взаимодействуют друг с другом?

Ссылка на видео этого руководства на сайте: borisproit.expert

В Android приложения взаимодействуют через **межпроцессное взаимодействие (IPC)**, потому что каждое приложение работает в **своём отдельном процессе и sandbox**.

Основные способы взаимодействия:

— **Intents** — запуск Activity, Service или отправка Broadcast
— **BroadcastReceiver** — получение системных или пользовательских событий
— **ContentProvider** — безопасный доступ к данным другого приложения
— **AIDL (Android Interface Definition Language)** — вызов методов между процессами
— **Binder** — низкоуровневый механизм IPC, на котором основаны многие системные сервисы

Чаще всего приложения взаимодействуют через **Intents и ContentProvider**, потому что это самый простой и безопасный способ.

How do applications interact with each other?

In Android, applications interact using **inter-process communication (IPC)** because each app runs in its own **process and sandbox**.

Main interaction mechanisms:

— **Intents** — start Activities, Services, or send broadcasts
— **BroadcastReceiver** — receive system or custom events
— **ContentProvider** — secure access to another app's data
— **AIDL (Android Interface Definition Language)** — call methods across processes
— **Binder** — the low-level IPC mechanism used by many system services

In practice, most app interactions happen through **Intents and ContentProviders** because they are the simplest and safest methods.

114. Что такое AIDL?

Что такое `AIDL`?

`AIDL (Android Interface Definition Language)` — это язык описания интерфейсов, который используется для **межпроцессного взаимодействия (IPC)** между приложениями или сервисами в Android.

AIDL позволяет одному процессу **вызывать методы в другом процессе**, как будто это обычный интерфейс.

Разработчик описывает интерфейс в `.aidl` файле, после чего Android генерирует код, который использует **Binder** для передачи данных между процессами.

Ссылка на видео этого руководства на сайте: borisproit.expert

---

What is `AIDL`?

`AIDL (Android Interface Definition Language)` is a language used to define interfaces for **inter-process communication (IPC)** between applications or services in Android.

AIDL allows one process to **call methods in another process** as if it were a regular interface.

Developers define the interface in an `.aidl` file, and Android generates the necessary code that uses **Binder** to transfer data between processes.

115. Когда используется AIDL?

Когда используется `AIDL`?

`AIDL` используется, когда нужно организовать **взаимодействие между разными процессами**, где один процесс предоставляет сервис, а другой вызывает его методы.

Это необходимо, когда простых механизмов (например `Intent`) недостаточно и требуется **вызов методов с передачей параметров и получением результата**.

Типичные случаи использования:

— взаимодействие **между двумя приложениями**
— работа с **удалённым сервисом (remote service)**
— создание **системных сервисов**
— когда нужно выполнять **многократные вызовы методов между процессами**

When is `AIDL` used?

`AIDL` is used when you need **communication between different processes**, where one process provides a service and another process calls its methods.

It is needed when simple mechanisms like `Intent` are not sufficient and you require **method calls with parameters and return values**.

Typical use cases:

— communication **between two applications**
— working with a **remote service**

Ссылка на видео этого руководства на сайте: borisproit.expert

— implementing **system services**
— when multiple **method calls between processes** are required

```
// ILocationService.aidl
interface ILocationService {
    Location getLastLocation();
}
```

App Process

↓

Binder IPC

↓

System Service Process

AIDL применяется, когда нужно **общаться между разными процессами**:

1. **Android system services**
    - LocationManager
    - ActivityManager
    - PackageManager
2. **Google Play Services**
3. **Связь между двумя приложениями**

Например:

- приложение-банк ↔ приложение-подписи
- приложение ↔ сервис платежей
- приложение ↔ VPN сервис

Ссылка на видео этого руководства на сайте: borisproit.expert

🟡 Broadcast deeper

117. Что такое sticky broadcast?

Что такое `sticky broadcast`?

`Sticky broadcast` — это специальный тип broadcast-сообщения, которое **сохраняется системой после отправки**.
Когда новый `BroadcastReceiver` регистрируется позже, он всё равно может **получить последнее отправленное сообщение**.

Это позволяло приложениям получать **последнее состояние системы**, даже если broadcast был отправлен раньше.

Однако sticky broadcasts **устарели (deprecated)** и не рекомендуются к использованию, потому что они могут создавать проблемы безопасности и непредсказуемое поведение.

What is a `sticky broadcast`?

A `sticky broadcast` is a special type of broadcast message that **remains stored in the system after being sent**.
When a new `BroadcastReceiver` registers later, it can still **receive the last broadcast that was sent**.

This allowed applications to retrieve the **latest system state**, even if the broadcast was sent earlier.

However, sticky broadcasts are **deprecated** and are no longer recommended because they can cause security issues and unpredictable behavior.

118. Почему sticky broadcast считается устаревшим?

Почему `sticky broadcast` считается устаревшим?

`Sticky broadcast` считается устаревшим, потому что он создаёт **проблемы безопасности и непредсказуемое поведение системы**.

Основные причины:

— **Проблемы безопасности** — любое приложение могло прочитать сохранённый broadcast и получить данные
— **Невозможно контролировать источник данных** — другое приложение могло

перезаписать broadcast
— **Глобальное состояние системы** — данные сохранялись на уровне системы и могли использоваться неправильно
— **Нарушение архитектуры Android** — механизм плохо вписывался в модель изолированных приложений

Поэтому Google объявил sticky broadcast устаревшим и рекомендует использовать другие механизмы, например **LiveData, Flow или обычные BroadcastReceiver**.

Why is `sticky broadcast` considered deprecated?

`Sticky broadcast` is deprecated because it introduces **security risks and unpredictable system behavior**.

Main reasons:

— **Security issues** — any application could read the stored broadcast and access its data
— **No control over the data source** — another app could overwrite the broadcast
— **Global shared state** — the broadcast remained stored system-wide and could be misused
— **Does not align with Android's app isolation model**

Because of these issues, Google deprecated sticky broadcasts and recommends using alternatives such as **LiveData, Flow, or regular BroadcastReceiver mechanisms**.

🟡 Sensors

119. Какие классы используются для работы с сенсорами в Android?

Какие классы используются для работы с сенсорами в Android?

Для работы с сенсорами в Android используется **Sensor Framework**.
Основные классы позволяют получать данные с датчиков устройства и реагировать на изменения.

Основные классы:

— `SensorManager` — главный класс для управления сенсорами и регистрации слушателей
— `Sensor` — представляет конкретный датчик (акселерометр, гироскоп, и т.д.)

Ссылка на видео этого руководства на сайте: borisproit.expert

— `SensorEvent` — содержит данные, полученные от сенсора
— `SensorEventListener` — интерфейс для получения обновлений от сенсора

Обычно приложение получает `SensorManager`, выбирает нужный `Sensor` и регистрирует `SensorEventListener`.

What classes are used to work with sensors in Android?

Android provides the **Sensor Framework** to work with device sensors.
The main classes allow applications to access sensor data and react to sensor changes.

Main classes:

— `SensorManager` — the main class used to manage sensors and register listeners
— `Sensor` — represents a specific sensor (accelerometer, gyroscope, etc.)
— `SensorEvent` — contains the data produced by a sensor
— `SensorEventListener` — interface used to receive sensor updates

Typically, an app obtains a `SensorManager`, selects a `Sensor`, and registers a `SensorEventListener`.

```
val sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager

val accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)

val listener = object : SensorEventListener {

    override fun onSensorChanged(event: SensorEvent) {
        // sensor data available here
    }

    override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {
        // accuracy changed
    }
}

sensorManager.registerListener(
    listener,
    accelerometer,
    SensorManager.SENSOR_DELAY_NORMAL
)
```

Ссылка на видео этого руководства на сайте: borisproit.expert

---

🟡 UI notifications

120. Что такое Toast?

Что такое `Toast`?

`Toast` — это небольшой всплывающий UI-элемент в Android, который используется для **краткого отображения сообщения пользователю**.
Он появляется на экране на короткое время и автоматически исчезает, не требуя взаимодействия пользователя.

`Toast` обычно используется для отображения **простых уведомлений или результатов действий**.

What is `Toast`?

A `Toast` is a small popup UI element in Android used to **display short messages to the user**.
It appears on the screen for a brief time and disappears automatically without requiring user interaction.

`Toast` is typically used for **simple notifications or action feedback**.

```
// Show a toast message
Toast.makeText(this, "Saved successfully", Toast.LENGTH_SHORT).show()
```

121. Когда использовать Toast вместо Dialog?

Когда использовать `Toast` вместо `Dialog`?

`Toast` используют, когда нужно показать **короткое информационное сообщение**, которое **не требует действий от пользователя**.
Он отображается на короткое время и автоматически исчезает.

`Dialog` используют, когда нужно **привлечь внимание пользователя и получить действие или подтверждение**.

Ссылка на видео этого руководства на сайте: borisproit.expert

`Toast` подходит для:
— простых уведомлений
— подтверждения действия (например, “Saved”)
— сообщений, которые не требуют взаимодействия

`Dialog` подходит для:
— подтверждения действий
— ошибок, требующих реакции пользователя
— ввода данных
— важных уведомлений

When should you use `Toast` instead of a `Dialog`?

`Toast` is used when you want to show a **short informational message** that **does not require user interaction**.
It appears briefly and disappears automatically.

A `Dialog` is used when you need to **get the user’s attention and require an action or confirmation**.

Use `Toast` for:
— simple notifications
— action confirmation (e.g., “Saved”)
— messages that do not require interaction

Use `Dialog` for:
— confirming actions
— errors requiring user response
— user input
— important notifications

🟡 Application lifecycle

122. Что такое класс Application?

Что такое класс `Application`?

`Application` — это базовый класс Android, который представляет **глобальное состояние приложения**.

Ссылка на видео этого руководства на сайте: borisproit.expert

Он создаётся **один раз при запуске процесса приложения** и существует до завершения процесса.

Класс `Application` используется для **инициализации глобальных компонентов**, которые должны жить столько же, сколько и всё приложение.

Типичные случаи использования:

— инициализация **Dependency Injection (Hilt, Dagger)**
— настройка **логирования или аналитики**
— инициализация **базы данных**
— создание **глобальных singleton объектов**

What is the `Application` class?

The `Application` class is a base Android class that represents the **global state of the application**.
It is created **once when the application process starts** and lives for the entire lifetime of that process.

The `Application` class is used to **initialize global components** that should live as long as the application itself.

Typical use cases:

— initializing **Dependency Injection (Hilt, Dagger)**
— configuring **logging or analytics**
— initializing the **database**
— creating **global singleton objects**

```
class MyApp : Application() {

    override fun onCreate() {
        super.onCreate()
        // App-wide initialization
    }
}
```

🟡 Build system

Ссылка на видео этого руководства на сайте: borisproit.expert

125. Что происходит во время build процесса Android приложения?

Что происходит во время build процесса Android приложения?

Во время build процесса исходный код приложения преобразуется в **установочный файл (APK или AAB)**, который можно установить на устройство.

Основные этапы build процесса:

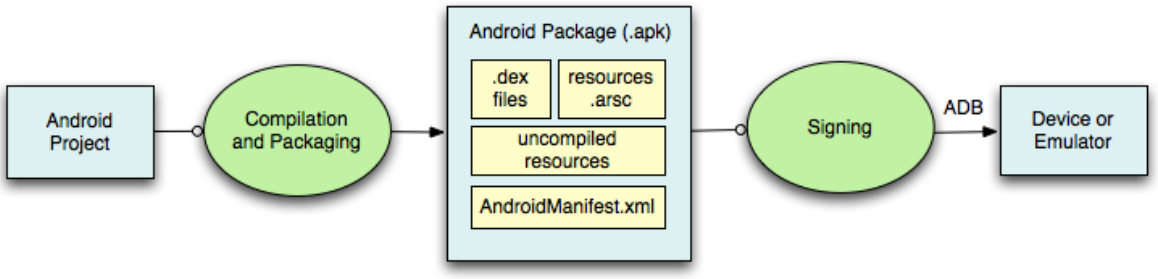
— **Компиляция кода** — Kotlin/Java код компилируется в **bytecode (.class)**
— **D8/R8 преобразование** — bytecode преобразуется в **DEX (Dalvik Executable)**, который может выполняться на Android
— **Обработка ресурсов** — XML layout, изображения и строки компилируются в бинарный формат
— **Минификация и обфускация** — R8 удаляет неиспользуемый код и переименовывает классы
— **Packaging** — код, ресурсы и манифест объединяются в **APK или AAB**
— **Подпись приложения** — приложение подписывается сертификатом разработчика

What happens during the **Android application build process**?

During the build process, the application's source code is transformed into an **installable package (APK or AAB)** that can run on a device.

Main build steps:

— **Code compilation** — Kotlin/Java code is compiled into **bytecode (.class)**
— **D8/R8 conversion** — bytecode is converted into **DEX (Dalvik Executable)** used by Android
— **Resource processing** — XML layouts, images, and strings are compiled into a binary format
— **Code shrinking and obfuscation** — R8 removes unused code and renames classes
— **Packaging** — code, resources, and manifest are bundled into an **APK or AAB**
— **App signing** — the application is signed with the developer's certificate



126. Что делает Gradle в Android проекте?

Ссылка на видео этого руководства на сайте: borisproit.expert

Что делает `Gradle` в Android проекте?

`Gradle` — это система сборки, которая **автоматизирует процесс build Android-приложения**.
Она управляет компиляцией кода, обработкой ресурсов, зависимостями и созданием APK или AAB.

Основные задачи Gradle:

— **управление зависимостями** (libraries)
— **компиляция Kotlin/Java кода**
— **обработка ресурсов** (layouts, strings, images)
— **запуск инструментов сборки** (D8, R8)
— **создание APK или AAB**
— **подпись приложения**
— **создание разных build variants** (debug, release)

Gradle также позволяет настраивать процесс сборки через **build.gradle файлы**.

What does `Gradle` do in an Android project?

`Gradle` is the build system used to **automate the Android application build process**.
It manages code compilation, resource processing, dependencies, and packaging the app into APK or AAB files.

Main responsibilities of Gradle:

— **dependency management** (libraries)
— **compiling Kotlin/Java code**
— **processing resources** (layouts, strings, images)
— **running build tools** (D8, R8)
— **building APK or AAB packages**
— **signing the application**
— **creating build variants** (debug, release)

Gradle also allows developers to configure the build process using **build.gradle files**.

127. Что такое build variants?

Что такое `build variants`?

`Build variants` — это разные **версии сборки одного и того же Android приложения**, которые создаются из одного проекта, но с разными настройками.

Ссылка на видео этого руководства на сайте: borisproit.expert

Они позволяют создавать, например, **debug и release версии**, которые могут отличаться конфигурацией, API-ключами, логированием или сервером.

`Build variant` формируется из комбинации **build type** и **product flavor`.

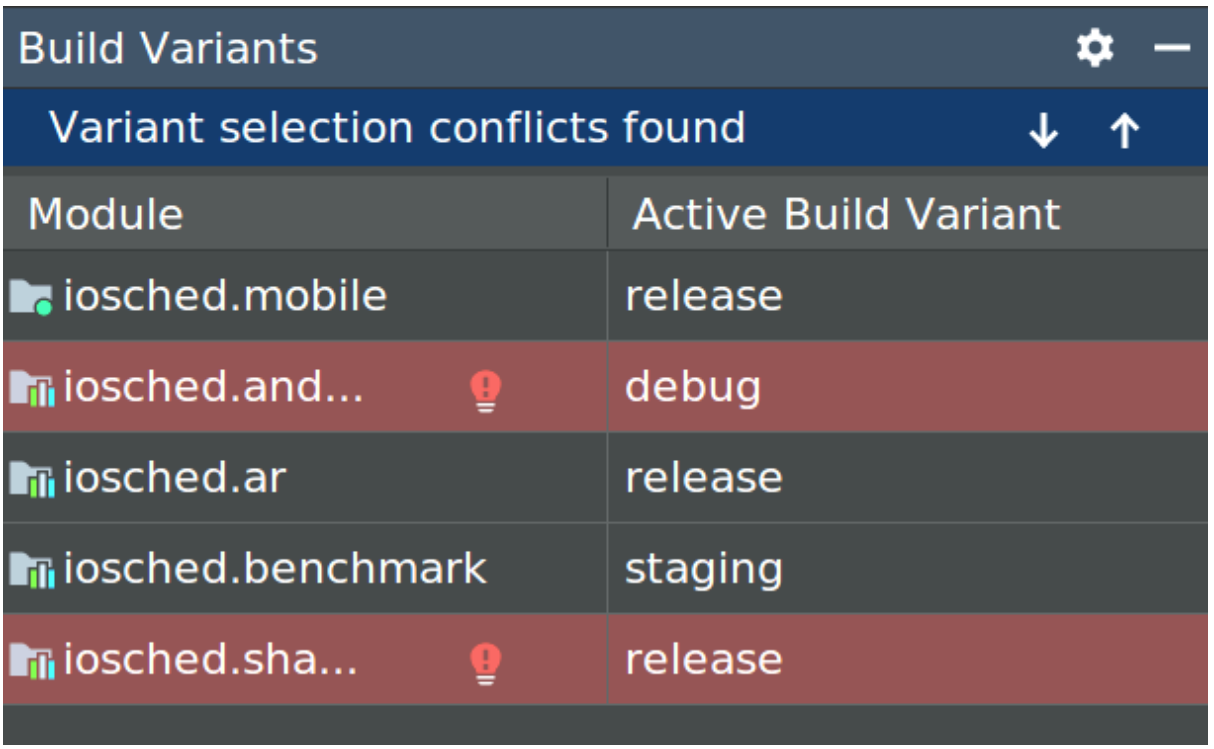
What are `build variants`?

`Build variants` are different **build versions of the same Android application** created from the same project but with different configurations.

They allow you to build, for example, **debug and release versions**, which may differ in configuration, API keys, logging, or backend server.

A `build variant` is created from a combination of **build type** and **product flavor**.

Ссылка на видео этого руководства на сайте: borisproit.expert



128. Чем debug build отличается от release build?

Чем `debug build` отличается от `release build`?

`Debug build` — это версия приложения, предназначенная для **разработки и тестирования**.
Она содержит инструменты отладки и не оптимизирована для финального использования.

`Release build` — это версия приложения, предназначенная для **публикации в Google Play**.
Она оптимизирована, подписана релизным ключом и не содержит инструментов отладки.

Основные отличия:

— `debug build` автоматически подписывается **debug key**, а `release build` — **release key**
— `debug build` поддерживает **debugging и logging**, в release это обычно отключено
— `release build` использует **R8/ProGuard для минификации и обфускации кода**
— `release build` оптимизирован для **меньшего размера и лучшей производительности**

What is the difference between a `debug build` and a `release build`?

Ссылка на видео этого руководства на сайте: borisproit.expert

A `debug build` is a version of the application intended for **development and testing**.
It includes debugging tools and is not optimized for final distribution.

A `release build` is the version of the application intended for **publishing on Google Play**.
It is optimized, signed with a release key, and does not include debugging tools.

Main differences:

— `debug build` is signed automatically with a **debug key**, while `release build` uses a **release key**
— `debug build` allows **debugging and logging**, which are usually disabled in release
— `release build` uses **R8/ProGuard for code shrinking and obfuscation**
— `release build` is optimized for **smaller size and better performance**

```
android {

    buildTypes {

        debug {
            isDebuggable = true
        }

        release {
            isMinifyEnabled = true
            isDebuggable = false
        }
    }
}
```

129. Что делает AAPT в Android build процессе?

Что делает `AAPT` в Android build процессе?

`AAPT (Android Asset Packaging Tool)` — это инструмент сборки Android, который отвечает за **обработку и упаковку ресурсов приложения**.

Он компилирует ресурсы (`XML`, изображения, строки), проверяет их корректность и генерирует вспомогательные файлы, которые используются в коде.

Основные задачи AAPT:

Ссылка на видео этого руководства на сайте: borisproit.expert

— компиляция ресурсов (`layouts`, `strings`, `drawables`)
— генерация класса **R**, который содержит идентификаторы ресурсов
— проверка ресурсов на ошибки
— упаковка ресурсов в APK или AAB

What does `AAPT` do in the Android build process?

`AAPT (Android Asset Packaging Tool)` is a build tool responsible for **processing and packaging application resources**.

It compiles resources such as `XML`, images, and strings, validates them, and generates helper files used in the code.

Main responsibilities of AAPT:

— compiling resources (`layouts`, `strings`, `drawables`)
— generating the **R class**, which contains resource identifiers
— validating resources for errors
— packaging resources into the APK or AAB

```
// Accessing resource generated by AAPT
val text = getString(R.string.app_name)
```

130. Что такое resource merging?

Что такое `resource merging`?

`Resource merging` — это процесс во время сборки Android-приложения, при котором **ресурсы из разных источников объединяются в один набор ресурсов**.

Android может получать ресурсы из нескольких мест:
— основного модуля приложения
— библиотек
— разных `build variants` (debug / release)
— `product flavors`

Во время сборки Gradle объединяет все ресурсы и решает конфликты, используя **приоритет источников**.

What is `resource merging`?

Ссылка на видео этого руководства на сайте: borisproit.expert

`Resource merging` is a build-time process in Android where **resources from multiple sources are combined into a single set of resources**.

Android resources can come from several places:
— the main application module
— libraries
— different `build variants` (debug / release)
— `product flavors`

During the build process, Gradle merges these resources and resolves conflicts using **a priority system**.

```
<!-- Main app resource -->
<color name="primaryColor">#FF0000</color>

<!-- Debug resource override -->
<color name="primaryColor">#00FF00</color>
```

🟡 Android threading primitives

131. Что такое Handler?

Что такое `Handler`?

`Handler` — это класс Android, который используется для **отправки и обработки сообщений или задач в конкретном потоке**, чаще всего в **Main (UI) thread**.

Он работает вместе с **Looper и MessageQueue**.
`Handler` помещает задачи (`Runnable`) или сообщения (`Message`) в очередь, и поток выполняет их последовательно.

Это позволяет **выполнять код в другом потоке или планировать выполнение задачи позже**.

What is `Handler`?

`Handler` is an Android class used to **send and process messages or tasks on a specific thread**, most commonly the **Main (UI) thread**.

Ссылка на видео этого руководства на сайте: borisproit.expert

It works together with **Looper and MessageQueue**.
A `Handler` posts tasks (`Runnable`) or messages (`Message`) into a queue, and the thread processes them sequentially.

This allows you to **execute code on another thread or schedule work to run later**.

```
// Hotel app example: load room details from database/network in background
// and update UI on the main thread using Handler

class RoomActivity : AppCompatActivity() {

    private val handler = Handler(Looper.getMainLooper())

    fun loadRoomDetails(roomId: Int) {

        Thread {

            // Simulate database or network request
            val room = "Room $roomId - Deluxe Suite"

            // Update UI on main thread
            handler.post {
                println("Room loaded: $room")
                // roomTitleTextView.text = room
            }

        }.start()
    }
}
```

Handler	Coroutines
сложный код	простой код
ручное управление потоками	автоматическое
легко сделать memory leak	lifecycle-aware
callback style	sequential style

132. Как работает Looper?

Как работает `Looper`?

Ссылка на видео этого руководства на сайте: borisproit.expert

`Looper` — это компонент Android, который **управляет циклом обработки сообщений в потоке**.
Он постоянно читает сообщения из **MessageQueue** и передаёт их `Handler`, который выполняет соответствующую задачу.

Каждый поток может иметь **один Looper**, который запускает бесконечный цикл обработки сообщений.

Основной поток приложения (`Main thread`) уже содержит `Looper`, поэтому `Handler` может использовать его для выполнения задач в UI потоке.

How does `Looper` work?

`Looper` is an Android component that **manages the message processing loop for a thread**.
It continuously reads messages from the **MessageQueue** and delivers them to a `Handler`, which executes the corresponding task.

Each thread can have **one Looper** that runs an infinite message-processing loop.

The main thread of an Android app already has a `Looper`, which allows `Handler` to post tasks to the UI thread.

Как это работает

В Android поток может иметь:

Thread

↓

Looper

↓

MessageQueue

↓

Handler

- **Handler** кладёт задачи в очередь
- **Looper** берёт задачи из очереди

- **Thread** выполняет их

133. Как Handler взаимодействует с MessageQueue?

Как `Handler` взаимодействует с `MessageQueue`?

`Handler` используется для **отправки сообщений или задач в очередь сообщений (`MessageQueue`) потока**.
Когда разработчик вызывает `post()` или `sendMessage()`, `Handler` помещает `Runnable` или `Message` в `MessageQueue`.

`Looper` постоянно читает элементы из `MessageQueue` и передаёт их обратно `Handler`, который выполняет соответствующий код.

Таким образом получается цепочка:

Handler → MessageQueue → Looper → Handler

How does `Handler` interact with `MessageQueue`?

A `Handler` is used to **send messages or tasks to a thread's `MessageQueue`**.
When a developer calls `post()` or `sendMessage()`, the `Handler` places a `Runnable` or `Message` into the `MessageQueue`.

The `Looper` continuously reads items from the `MessageQueue` and delivers them back to the `Handler`, which executes the corresponding code.

So the flow looks like this:

Handler → MessageQueue → Looper → Handler

🟡 Testing basics

134. Какие виды тестирования есть в Android?

Какие виды тестирования есть в Android?

Ссылка на видео этого руководства на сайте: borisproit.expert

В Android обычно выделяют **три основных уровня тестирования**, которые проверяют разные части приложения.

Основные виды тестирования:

— **Unit tests** — проверяют отдельные классы или функции без Android framework
— **Integration tests** — проверяют взаимодействие нескольких компонентов (например Repository + Database)
— **UI tests** — проверяют пользовательский интерфейс и взаимодействие пользователя с приложением

Unit tests обычно выполняются на **JVM**, а UI tests — на **эмуляторе или реальном устройстве**.

What types of testing exist in Android?

In Android development, testing is usually divided into **three main levels**, each verifying different parts of the application.

Main testing types:

— **Unit tests** — test individual classes or functions without the Android framework
— **Integration tests** — test interactions between multiple components (for example Repository + Database)
— **UI tests** — test the user interface and user interactions with the app

Unit tests typically run on the **JVM**, while UI tests run on an **emulator or a real device**.

135. Что такое unit test в Android?

Что такое `unit test` в Android?

`Unit test` — это тест, который проверяет **работу отдельного класса или функции изолированно**, без зависимости от Android framework, базы данных или сети.

Цель unit test — убедиться, что **бизнес-логика работает правильно** при разных входных данных.

Такие тесты обычно запускаются **на JVM**, поэтому они выполняются быстро и не требуют эмулятора или устройства.

What is a `unit test` in Android?

Ссылка на видео этого руководства на сайте: borisproit.expert

A `unit test` is a test that verifies **a single class or function in isolation**, without relying on the Android framework, databases, or network.

The goal of a unit test is to ensure that the **business logic behaves correctly** for different inputs.

These tests usually run **on the JVM**, so they are fast and do not require an emulator or a physical device.

```
class Calculator {

    fun sum(a: Int, b: Int): Int {
        return a + b
    }
}

class CalculatorTest {

    @Test
    fun sum_returnsCorrectValue() {
        val calculator = Calculator()

        val result = calculator.sum(2, 3)

        assertEquals(5, result)
    }
}
```

136. Что такое instrumentation test?

Что такое `instrumentation test`?

`Instrumentation test` — это тест, который выполняется **на реальном устройстве или эмуляторе** и может взаимодействовать с **Android framework**.

Такие тесты используются для проверки компонентов Android, например `Activity`, `Fragment`, `UI`, базы данных или взаимодействия между компонентами.

Instrumentation tests запускаются через **AndroidJUnitRunner** и имеют доступ к `Context` и другим Android API.

Ссылка на видео этого руководства на сайте: borisproit.expert

---

What is an `instrumentation test`?

An `instrumentation test` is a test that runs **on a real device or emulator** and can interact with the **Android framework**.

These tests are used to verify Android components such as `Activity`, `Fragment`, `UI`, databases, or interactions between components.

Instrumentation tests run using **AndroidJUnitRunner** and have access to `Context` and other Android APIs.

```
@RunWith(AndroidJUnit4::class)
class MainActivityTest {

    @Test
    fun activity_launchesSuccessfully() {
        val scenario = ActivityScenario.launch(MainActivity::class.java)

        assertNotNull(scenario)
    }
}
```

137. Чем unit test отличается от instrumentation test?

Чем `unit test` отличается от `instrumentation test`?

`Unit test` проверяет **отдельную часть логики приложения** (например функцию или класс) **без Android framework**.
Он запускается **на JVM**, поэтому работает быстро и не требует устройства или эмулятора.

`Instrumentation test` проверяет **Android компоненты и их взаимодействие**, например `Activity`, `Fragment`, UI или работу с `Context`.
Он запускается **на устройстве или эмуляторе** и имеет доступ к Android framework.

Главные отличия:

— `unit test` выполняется на **JVM**, `instrumentation test` — **на устройстве/эмуляторе**
— `unit test` **не использует Android framework**, `instrumentation test` **использует**
— `unit test` работает **быстро**, `instrumentation test` **медленнее**

Ссылка на видео этого руководства на сайте: borisproit.expert

— `unit test` проверяет **бизнес-логику**, `instrumentation test` проверяет **Android компоненты и UI**

What is the difference between a `unit test` and an `instrumentation test`?

A `unit test` verifies **a small unit of application logic**, such as a function or class, **without using the Android framework**.
It runs **on the JVM**, so it is fast and does not require a device or emulator.

An `instrumentation test` verifies **Android components and their interactions**, such as `Activity`, `Fragment`, UI, or `Context`.
It runs **on a device or emulator** and has access to the Android framework.

Main differences:

— `unit test` runs on the **JVM**, while `instrumentation test` runs on a **device/emulator**
— `unit test` **does not use Android framework**, `instrumentation test` **does**
— `unit test` is **fast**, `instrumentation test` is **slower**
— `unit test` tests **business logic**, `instrumentation test` tests **Android components and UI**