Kotlin

var/val - мутабельная и иммутабельная переменная соответственно.
val можно изменять только через состояние объекта.

const val/companion object - статические переменные, доступ к которым можно получить не имея объект класса.
const val - compile-time константа (аналог final static в JVM).
val и companion object - runtime константа.

data class - это обертка над обычным классом, которая упрощает работу с некоторыми из них, которые предназначены для хранения данных. Data class имеют свои переопределенные методы - *toString, equals, hashcode, copy, componentN.*
Если переопределить хотя бы один из *toString, equals, hashcode, copy* - остальные не сгенерируются автоматически.
Эти методы генерируются только по свойствам из основного (primary) конструктора.
Объявить **var** в **primary constructor data class** можно, но плохо: equals()/hashCode() зависят от изменяемого поля → объект может "потеряться" в Set/Map после мутации.

Деструктуризация - это фича Kotlin, которая позволяет «распаковать» объект сразу в несколько переменных. Самый частый и понятный пример — с data class (component1(), component2() и т.д.).

```
// Деструктуризация:
val (name, age, email) = user
```

Обычные классы тоже могут использовать деструктуризацию, если переопределить все *operator fun component1() = x;* в конструкторе напрямую.

Inline usecases
Модификатор **inline** заставляет компилятор Kotlin (не JVM) встраивать тело функции прямо в место вызова — то есть вместо вызова функции подставляется её код.
Это происходит на этапе компиляции Kotlin → байт-код, а не на уровне JVM.
Зачем это нужно?
Главное преимущество — устранение оверхеда при работе с лямбдами.
Без inline лямбда превращается в объект (создаётся анонимный класс), что дорого по памяти и времени.
С inline — лямбда встраивается как обычный код, и объекта не создаётся.

Когда **inline** используют почти всегда:

- let, apply, also, run, with — все они inline
- Любая функция, принимающая лямбду как параметр и вызывающая её напрямую
- Особенно важно для high-order functions

Когда **inline** НЕЛЬЗЯ использовать:

- Если функция содержит return из внешней функции (обычно запрещено)
- Если функция большая — инлайнинг раздувает код

Бонус: noinline и crossinline

- **crossinline** — если внутри инлайн-функции лямбда передаётся дальше (например, в другую функцию), и нельзя делать non-local return
- **noinline** — если одну из лямбд не хочешь инлайнить

Когда использовать **crossinline**:

- Лямбда передаётся в другую функцию (в т.ч. в run, launch, async, Thread, setOnClickListener и т.д.)
- Ты хочешь запретить return из лямбды (чтобы не сломать логику внешней функции)

Когда использовать **noinline**:
У inline-функции все лямбды по умолчанию инлайнятся.
Но иногда одну из лямбд нельзя или не хочется инлайнить.
Случай 1: Лямбда сохраняется как объект (например, в свойство)
Случай 2: Лямбда слишком большая — не хочется раздувать код

Оператор **!!** - форсирует компилятор воспринимать переменную как nonnullable.
!! - бросает KotlinNullPointerException, если значение null.

== в Kotlin — это структурное сравнение (вызывает .equals())
=== это сравнение ссылок (то же, что == в Java).

Extension function — это функция, которая добавляет поведение к существующему классу без наследования и изменения его кода.
Паттерн проектирования, который реализуют extension functions — Decorator (в открытой форме) или чаще говорят Visitor, но в сообществе Kotlin чаще всего называют это реализацией паттерна Extension / Open/Closed Principle (открыт для расширения, закрыт для модификации).

Главное преимущество **sealed class** — исчерпывающая проверка в when (exhaustive when).
Компилятор обязывает обработать все возможные подклассы, иначе — ошибка компиляции.

Enum тоже это делает, но:

- enum — только фиксированные экземпляры (один объект на значение)
- sealed class — каждый подтип может иметь разные свойства и состояние (как обычные классы)

Когда обязательно использовать sealed class вместо обычного:
Когда ты хочешь моделировать ограниченное множество типов (как Result, ViewState, UiEvent и т.д.) и нужна исчерпывающая обработка в when + каждое состояние может нести свои данные.
Пример:

```
sealed class Result {
    data class Success(val data: String) : Result()
    data class Error(val exception: Throwable) : Result()
    object Loading : Result()
}
```

Any — корень всей иерархии типов Kotlin (как Object в Java). Все классы неявно наследуются от Any. Не-nullable по умолчанию.
Unit — тип, обозначающий «функция ничего полезного не возвращает». Имеет единственное значение — Unit (объект-одиночка). Аналог void в Java, но является настоящим типом. Пример: fun printHello(): Unit { println("Hello") } (можно опустить : Unit)
Nothing — тип, у которого нет значений. Означает «функция никогда не завершится нормально». Используется для:

- функций, которые всегда бросают исключение: fun fail(): Nothing = throw RuntimeException()
- бесконечных циклов
- мест, где компилятор понимает, что код недостижим

Nothing? — nullable версия Nothing. Единственное возможное значение — null. Полезен в generics для обозначения «список, который всегда пуст»: List<Nothing?>

Только **const val** — настоящая compile-time константа, которую можно использовать в аннотациях.
@JvmStatic val в companion — даёт статическое поле + статический геттер, удобно вызывать из Java как MyClass.VALUE.
@JvmField val — убирает геттер полностью, поле становится как обычное public поле Java. Часто используют в data class-ах, когда нужно совместимость с фреймворками (Gson, Jackson и т.д.).

По умолчанию в **Kotlin**:

- Все классы — final (нельзя наследоваться).
- Все функции и свойства — final (нельзя переопределять).

open class / open fun:

- open class — разрешает наследование от этого класса.
- open fun / open val — разрешает переопределять эту функцию или свойство в дочерних классах.
- Используется редко — только когда ты явно хочешь разрешить переопределение.

abstract class / abstract fun:

- abstract class — нельзя создать объект напрямую, обязательно нужно унаследовать и реализовать абстрактные члены.
- abstract fun / abstract val — не имеют тела/значения, обязательно переопределять в наследнике.
- Может содержать и обычные функции с реализацией, и состояние (поля).
- Используется, когда есть общее поведение + нужно заставить наследников реализовать что-то своё.

interface:

- По умолчанию все функции — public open (можно переопределять, но не обязательно).
- С Kotlin 1.2+ можно писать функции с телом (реализация по умолчанию).
- До Kotlin 1.4 нельзя было иметь поля с бэкингом (только геттеры/сеттеры), сейчас можно только private val.
- Класс может реализовывать сколько угодно интерфейсов.
- Используется для описания поведения (контракта), особенно когда нужна множественная реализация.

Краткое правило выбора:

- Нужна множественная реализация → interface
- Есть общее состояние или защищённые члены → abstract class
- Хочешь запретить наследование по умолчанию → просто class (final)
- Редко: хочешь контролируемо разрешить наследование → open class + open fun

public

- Везде и всегда видно: из любого файла, пакета, модуля.
- Это модификатор по умолчанию (если ничего не написать).

internal

- Видно только внутри одного модуля (в Gradle — один проект = один модуль).
- Не зависит от пакета: даже если классы в разных пакетах, но в одном модуле — видно друг другу.
- Если вынесешь в другой модуль — станет невидимым.
- Главное отличие от Java package-private.

protected

- Нельзя использовать на top-level (вне класса).
- Внутри класса: видно в этом классе и во всех его подклассах, даже если подкласс в другом пакете или модуле.
- Не видно из других классов, даже в том же пакете.

private

- Top-level (вне класса): видно только внутри одного .kt файла.
- Внутри класса: видно только внутри этого класса, подклассы не видят.
- Самый строгий уровень.

В обычных generic-функциях тип стирается на этапе выполнения (type erasure) — в JVM в рантайме нет информации, какой именно T был передан.

```
fun <T> createList(): List<T> = ArrayList<T>()  // в рантайме T = Unknown
```

reified + inline позволяет сохранить реальный тип в рантайме, но только внутри inline-функций.

```
inline fun <reified T> Any.isInstanceOf(): Boolean = this is T
```

reified — это суперсила Kotlin, которая позволяет писать type-safe, чистый и удобный API для работы с дженериками в рантайме.
Без inline fun <reified T> таких вещей, как filterIsInstance(), Gson.fromJson<T>(), Room @Query, Moshi.adapter<T>() — просто бы не существовало в удобной форме.
Использовать этот модификатор нужно именно для рефлексии и всего, что с ней связано. В рантайме (в JVM) обычные дженерики стираются, поэтому без reified ты не знаешь, какой именно T был передан.

- Проверка типа (is, as)
- Получение Class<T> или KClass<T>
- Работа с аннотациями на типе
- Сериализация/десериализация (Gson, Moshi, Jackson-Kotlin, Room)
- Room @Query с generic DAO:
- Стандартные функции (filterIsInstance()<T>, intent.getSerializableExtra<T>())

Bytecode — это файл/последовательность инструкций (в формате .class), которые получаются после компиляции Java/Kotlin-кода. Это статический набор команд для виртуальной машины Java.
JVM (runtime) — это программа-исполнитель, которая в момент запуска читает и выполняет этот bytecode. Это динамическая среда выполнения: загрузчик классов, куча, сборщик мусора, JIT-компилятор, стеки потоков и т.д.

Generics в Kotlin (и Java) нужны для трёх главных вещей:

1. Type-safety на этапе компиляции

```
val list: List<String> = listOf("hello", "world")
val s: String = list[1]  // OK, компилятор знает, что там String
// list.add(123) — ошибка компиляции!
```

2. Избавление от лишних приведений типов (casts)

```
// Java-style (до generics)
String s = (String) list.get(0);
// Kotlin с generics — каст не нужен
val s: String = list[0]  // автоматически
```

3. Повторное использование кода с разными типами. Один и тот же класс/функция работает с любым типом:

```
fun <T> identity(value: T): T = value
identity("hello")   // T = String
identity(42)        // T = Int
identity<User>(user)
```

Инструкция (**statement**) — делает что-то, но не возвращает значение
Выражение (**expression**) — всегда возвращает значение и имеет тип
В Kotlin почти всё — выражение:

- if — выражение (обязательно должен быть else, если используется как выражение)
- when — выражение
- try-catch — выражение
- Даже elvis ?: throw — выражение

В Java if, for, while — только инструкции → нельзя присвоить результат.

1. **Int** — примитивный тип в Kotlin (на уровне JVM — int). Не может быть null, не является объектом, не наследуется от Any?.
2. **Int?** — nullable примитив, в JVM при использовании — автоматически боксится в java.lang.Integer.
3. **kotlin.Int** — это алиас (то же самое, что просто Int). Пишется полностью только в редких случаях (например, при рефлексии).
4. **java.lang.Integer** — обёртка из Java. Всегда объект, может быть null, наследуется от Any?.

Главный вопрос: почему Int НЕ наследуется от Any??
Потому что Int — примитив (int), а примитивы в JVM не участвуют в иерархии объектов. Any в Kotlin соответствует java.lang.Object → только ссылочные типы наследуются от него.

vararg — синтаксический сахар для передачи нефиксированного количества аргументов одного типа в функцию. Под капотом превращается в Array<T>.
List<T> — интерфейс коллекции (immutable или mutable). Нельзя использовать как параметр функции вместо vararg.
Array<T> — примитивный массив (как int[] в Java). Это не List, у него другая производительность и поведение.

Почему for (i in list) работает?
Потому что у List<T> есть расширение-оператор **iterator**():

```
operator fun <T> Iterable<T>.iterator(): Iterator<T>
```

Под капотом:

- 0..9 → IntRange → наследует IntProgression
- Все эти .., until, downTo, step — возвращают объекты типа IntProgression

Диапазоны в Kotlin — это не просто синтаксический сахар, а настоящие объекты.
Оператор **..** создаёт включительный диапазон: от начального значения до конечного, включая оба края.
Функция until создаёт исключающий верхнюю границу диапазон: конечное значение в цикл не попадает.
downTo создаёт диапазон, идущий вниз, и конечная граница включается.
step позволяет задать шаг больше 1, работает и в прямом, и в обратном направлении.
until спасает от классической off-by-one ошибки: когда нужно пройти по индексам списка или массива, правильнее писать от нуля и до size с until, а не с .., потому что последний индекс всегда size - 1.
Диапазоны работают не только с числами: можно перебирать символы от 'a' до 'z' — это тоже полноценный диапазон.
Цикл for в Kotlin на самом деле работает не только с диапазонами, а с любым объектом, у которого определён оператор iterator. У всех стандартных коллекций (List, Set, Map.keys, Map.values и т.д.) этот оператор есть по умолчанию, поэтому по любой коллекции можно писать for (item in collection) — это обычный foreach.
Под капотом диапазоны .., until, downTo, step реализуются через класс IntProgression (или LongProgression, CharProgression). Это лёгкие объекты с тремя полями: start, end и step. Они не создают промежуточных списков — перебор идёт эффективно, почти как обычный цикл while.

Все виды **object** в Kotlin:

1. **Object declaration** (именованный объект, **синглтон**) Это настоящий синглтон — в приложении существует ровно один экземпляр этого объекта. Создаётся лениво при первом обращении. Используется вместо статических полей/методов из Java, для логгеров, конфигов, кэшей и т.д. Пример из жизни: Json, Clock.System, Dispatchers.Main.
2. **Companion object** Это тот же синглтон, но живёт внутри класса и привязан к нему. Используется как место для фабричных методов, констант и всего, что в Java было бы static. У каждого класса может быть максимум один companion object (с именем или без). В стандартной библиотеке — у всех коллекций: List, Map, Set и т.д.
3. **Object expression** (анонимный объект) Это не синглтон! Каждый раз создаётся новый экземпляр. Это прямой аналог анонимного класса в Java. Используется там, где нужно быстро реализовать интерфейс или унаследовать класс в одном месте: клик-листенеры, коллбэки, обработчики событий и т.д.
4. Вложенные **object внутри object** Можно внутри одного синглтона объявлять другие объекты. Используется для красивой группировки констант и подмодулей: например, Retrofit, Endpoints, Headers, Permissions и т.д.

Primary constructor — это часть заголовка класса: class User(val name: String). Он выполняется до любого init-блока.
Secondary constructor — это обычный constructor(...) внутри тела класса, вызывается через this(...) и обязателен только если нет primary или нужно несколько вариантов создания.

Да, **можно расширять любой класс** (даже final и даже из Java) через extension functions/properties. Это одна из главных фишек Kotlin.

by lazy — ленивая инициализация при первом обращении, значение кэшируется, работает с val.
lateinit var — инициализация вручную позже, только для var и только для классов (не примитивов), иначе краш.

@JvmStatic в companion object делает метод/свойство настоящим static в байткоде. Без него из Java вызываешь MyClass.Companion.method(), с ним — MyClass.method().

Delegates.observable(initialValue) { prop, old, new -> … } — создаёт свойство, которое вызывает лямбду при каждом изменении значения.

by lazy по умолчанию использует **LazyThreadSafetyMode.SYNCHRONIZED** — потокобезопасно с блокировкой.
by lazy(**LazyThreadSafetyMode.NONE**) { … } — без синхронизации, быстрее, но только если уверен, что доступ из одного потока.

typealias — псевдоним типа.
Очень полезен для длинных функциональных типов:
typealias ClickListener = (View) -> Unit

value class (с Kotlin 1.5+, раньше inline class) — обёртка над одним значением (примитивом), которая не создаёт объект в рантайме при использовании.
Пример: @JvmInline value class UserId(val value: Long) — в JVM будет просто long.

expect / actual — механизм multiplatform.
expect class Platform в commonMain, actual class Platform в androidMain/iosMain/jvmMain и т.д.
expect = «я обещаю, что где-то будет реализация»
actual = «вот реализация именно для этой платформы»

let — возвращает результат лямбды, it-референс, часто для цепочек и null-check.
apply — возвращает сам объект, this-референс, для настройки объекта.
run — возвращает результат лямбды, this-референс, для выполнения блока и получения результата.

TODO() — функция, всегда бросает NotImplementedError, тип Nothing.

tailrec — оптимизирует tail-recursive функцию в цикл, предотвращая StackOverflow.

by — делегирование свойства в Kotlin.
Свойство не хранит значение само, а передаёт геттер/сеттер другому объекту (делегату), реализующему интерфейс ReadOnlyProperty или ReadWriteProperty.
Как работает:

1. Компилятор генерирует скрытое поле для хранения делегата.
2. При обращении к свойству вызывает getValue() / setValue() у делегата.
3. Делегат может вычислять значение, кэшировать, логировать и т.д.

Invariance: Default in generics. List<String> не подтип List<Any>.
Обеспечивает максимальную безопасность, но ограничивает гибкость.
List<String> нельзя присвоить переменной List<Any> и наоборот.
Коллекции, которые можно изменять (например, MutableList<T>), по умолчанию инвариантны.
Covariance (out T): Producer-вариант. List<out String> подтип List<out Any>.
Позволяет безопасно использовать более "специфичные" типы в контексте "более общих". Идеально для только для чтения (read-only) коллекций или интерфейсов, которые возвращают параметр T.
List<String> это List<Any> (безопасно, так как String - это Any, но нельзя добавить в List<Any> что-то кроме Any).
Contravariance (in T): Consumer-вариант. Comparator<in Any> подтип Comparator<in String>.
Позволяет использовать более "общие" типы там, где ожидаются "специфичные", если интерфейс только принимает параметр T (например, функция, которая принимает T).
Comparator<Any> может сравнивать Int и Double, потому что Any "шире", чем Int или Double

List<*> — unknown generic (star-projection): тип неизвестен, можно читать как Any?, нельзя писать ничего.
List<Any?> — конкретный тип: элементы Any? (nullable Any), можно писать null или любой объект, читать как Any?.
List<*> безопаснее для чтения из неизвестных списков, без кастов.

inner class — вложенный класс, который имеет ссылку на внешний класс (non-static в Java).

- Доступен this@Outer из inner.
- Можно обращаться к свойствам/методам внешнего класса напрямую.
- Нельзя создать inner без экземпляра outer.

Без inner — просто nested class (static, без ссылки на outer).

Композиция — "has-a" отношение: класс содержит экземпляр другого класса как свойство.

```
class Engine(val power: Int)
class Car {
    private val engine = Engine(200)  // композиция: Car имеет Engine
    fun start() = println("Engine $engine.power started")
}
```

Car использует Engine, а не наследуется от него.
Основной принцип ООР: предпочитай композицию наследованию (гибче, меньше связанности).

ОП, ООП

Value type vs reference type:
Value type хранит значение напрямую в стеке (примитивы как int, boolean).
Reference type хранит ссылку на объект в куче.
Копирование value type копирует значение, reference — только ссылку.

Функция vs метод:
Функция — самостоятельный блок кода, не привязан к классу.
Метод — функция внутри класса, может работать с его полями и вызываться на объекте.
В Kotlin всё обычно методы, но top-level функции — как функции.

Рекурсия:
Функция вызывает сама себя для решения задачи на подзадачах.
Полезна для деревьев, факториалов, но рискует переполнить стек.
Требует базового случая для остановки.

ООП:
Парадигма, где программа строится из объектов, сочетающих данные и поведение.
Ключевые принципы: инкапсуляция, наследование, полиморфизм, абстракция.
Позволяет моделировать реальный мир.

Инкапсуляция:
Инкапсуляция в объектно-ориентированном программировании представляет собой принцип сокрытия внутренних деталей реализации класса от внешнего мира, обеспечивая доступ к данным только через контролируемые методы (геттеры и сеттеры). Это повышает безопасность и упрощает поддержку кода. В Android-разработке примером служит ViewModel, где поля состояния объявляются private, а доступ к ним предоставляется через public-методы или observable свойства (LiveData/StateFlow), предотвращая прямую модификацию данных из Activity или Fragment и обеспечивая контролируемые обновления UI.

Наследование:
Наследование в ООП — это механизм, позволяющий одному классу (подклассу) получать свойства и методы другого класса (суперкласса), способствуя переиспользованию кода и созданию иерархий. В Kotlin классы и методы по умолчанию final, поэтому для наследования требуется модификатор open. Это позволяет расширять функциональность базового класса без дублирования кода.

Полиморфизм:
Полиморфизм в ООП подразумевает способность объектов разных классов обрабатываться через общий интерфейс или суперкласс, проявляясь в overriding (динамический полиморфизм) или overloading. Пример в Android: разные реализации Repository (RoomRepository, NetworkRepository) могут реализовывать один интерфейс Repository, позволяя ViewModel работать с любой реализацией без изменений, что упрощает переключение источников данных. Включает runtime (overriding) и compile-time (overloading).
Overloading:
Несколько методов с одним именем, но разными параметрами.
Разрешается на этапе компиляции.
Удобно для разных способов вызова.
Overriding:
Переопределение метода родителя в потомке.
Работает в runtime (dynamic dispatch).
Требует open в родителе.

Абстракция:
Абстракция в ООП — принцип скрытия сложных деталей реализации и предоставления только необходимого интерфейса для взаимодействия. Это достигается через абстрактные классы или интерфейсы, фокусируя внимание на "что" делает объект, а не "как". Абстракция упрощает проектирование систем и повышает читаемость кода.

Разница между абстрактным и обычным классом заключается в том, что абстрактный класс не может быть инстанциирован напрямую и может содержать абстрактные методы (без реализации), которые должны быть переопределены в подклассах, в то время как обычный класс полностью реализован и готов к созданию объектов.

Абстрактный класс vs интерфейс:
Абстрактный класс — частичная реализация + состояние.
Интерфейс — контракт, с Kotlin 1.2+ реализация по умолчанию, без состояния (кроме private).
Класс наследует один абстрактный, реализует много интерфейсов.

Интерфейс
Интерфейс в ООП — это контракт, определяющий набор методов, которые класс должен реализовать, без хранения состояния (кроме private полей в Kotlin). Разница с абстрактным классом: интерфейс поддерживает множественную реализацию, может иметь default-реализации, но не наследует состояние; абстрактный класс — единственное наследование с возможным состоянием и частичной реализацией.
В Android интерфейсы предпочитают для callbacks, поскольку они позволяют множественную реализацию, упрощают тестирование (легко мокать) и избегают проблем наследования, обеспечивая слабую связанность между компонентами (например, OnClickListener).

Композиция предпочтительнее наследования в Android, поскольку она обеспечивает большую гибкость, снижает связанность и избегает проблем "хрупкой базовой иерархии". Пример: ViewModel содержит экземпляр Repository как поле (has-a отношение), позволяя легко заменить реализацию (например, на mock для тестов) без изменения иерархии классов.

ООП применяется в Android компонентах через наследование (Activity наследует AppCompatActivity), инкапсуляцию (private поля в Fragment), полиморфизм (разные View в RecyclerView) и абстракцию (интерфейсы для callbacks), что позволяет создавать расширяемые и тестируемые компоненты с чётким разделением ответственности.

Проблемы множественного наследования в Android избегают использованием интерфейсов вместо классов для контрактов, а также композицией и делегированием, что предотвращает конфликты методов и упрощает архитектуру.

SOLID:
Принципы для чистого, поддерживаемого кода.

Single Responsibility Principle:
Класс имеет одну причину для изменения (одна ответственность).
Упрощает тестирование и поддержку.
Пример SRP в Android:
ViewModel только управляет данными, Repository — только доступ к БД.

Open/Closed Principle:
Открыт для расширения, закрыт для модификации.
Пример OCP в Android:
Расширяем RecyclerView.Adapter новыми ViewHolder без изменения базового.

Liskov Substitution Principle:
Подтипы заменяемы на супер-типы без нарушения поведения.
Пример LSP в Android:
CustomView наследует View и работает везде, где ожидается View.

Interface Segregation Principle:
Много маленьких интерфейсов лучше одного большого.
Пример ISP в Android:
Отдельные интерфейсы для click/listener вместо одного огромного.

Dependency Inversion Principle:
Зависеть от абстракций, не от конкретных реализаций.
Пример DIP в Android:
ViewModel зависит от Repository интерфейса, а не конкретного RoomRepo; инжект через Hilt.

Factory pattern:
Создаёт объекты без указания конкретного класса.
Пример Factory в Android:
ViewModelProvider.Factory для создания ViewModel с параметрами.

Builder pattern:
Пошаговое создание сложного объекта.
Пример Builder в Android:
AlertDialog.Builder для настройки диалога.

Observer pattern:
Объекты уведомляют подписчиков об изменениях.
Пример Observer в Android:
LiveData/StateFlow — View наблюдает за данными в ViewModel.

Adapter pattern:
Преобразует интерфейс одного класса в другой.
Пример Adapter в Android:
RecyclerView.Adapter адаптирует данные для ViewHolder.

Facade pattern:
Упрощённый интерфейс к сложной подсистеме.
Пример Facade в Android:
MediaPlayer или Retrofit скрывают сложность.

Proxy pattern:
Контролирует доступ к объекту (lazy, cache).
Пример Proxy в Android:
Glide/Picasso для ленивой загрузки изображений.

Command pattern:
Инкапсулирует запрос как объект.
Пример Command в Android:
Undo/Redo операции в редакторах
.
Strategy pattern:
Сменяемая стратегия поведения.
Пример Strategy в Android:
Разные сортировки в RecyclerView DiffUtil.

State pattern:
Поведение меняется в зависимости от состояния.
Пример State in Android UI:
Loading/Error/Success экраны в ViewModel.

MVC:
Model-View-Controller: Model данные, View UI, Controller логика.
Как MVC применяется в Android:
Activity/Fragment как Controller+View, Model отдельно.

MVP:
Model-View-Presenter: Presenter посредник.
Как MVP применяется в Android:
Presenter держит логику, View — интерфейс с методами.

MVVM:
Model-View-ViewModel: ViewModel экспонирует данные.
Как MVVM применяется в Android:
ViewModel + LiveData/StateFlow + DataBinding.

Разница MVC/MVP/MVVM:

Эти три архитектуры предназначены для разделения ответственности в приложении, чтобы код был более тестируемым, поддерживаемым и масштабируемым. В Android они применяются для организации взаимодействия между UI, логикой и данными. Ниже — детальное описание каждой и сравнение.

MVC (Model-View-Controller)

- **Model**: отвечает за данные и бизнес-логику (репозитории, базы данных, сетевые запросы).
- **View**: отвечает за отображение (Activity/Fragment, XML-лейауты).
- **Controller**: посредник — обрабатывает пользовательский ввод, обновляет Model и View.
- **Как работает в Android**: Activity/Fragment часто выступает одновременно Controller и View (Controller получает события, обновляет Model, вручную обновляет UI).
- **Плюсы**: простота, подходит для маленьких приложений.
- **Минусы**: Activity становится "жирной" (God object), сложно тестировать, сильная связанность View и Controller.

MVP (Model-View-Presenter)

- **Model**: те же данные и бизнес-логика.
- **View**: пассивный интерфейс — только отображает данные и передаёт события (интерфейс с методами showProgress(), showData()).
- **Presenter**: вся логика приложения — получает события от View, работает с Model, обновляет View через интерфейс.
- **Как работает в Android**: Activity/Fragment реализует View-интерфейс, Presenter держит ссылку на View. Нет прямой зависимости от Android-фреймворка в Presenter.
- **Плюсы**: Presenter легко unit-тестировать (без Android), чёткое разделение, View пассивен.

- **Минусы**: много boilerplate-кода (интерфейсы, ручное обновление View), Presenter может стать большим, проблемы с lifecycle (нужно отписываться вручную).

MVVM (Model-View-ViewModel)

- **Model**: данные и репозитории.
- **View**: UI (Activity/Fragment/Compose), наблюдает за данными.
- **ViewModel**: хранит и управляет UI-состоянием, экспонирует observable данные (LiveData/StateFlow), работает с Model.
- **Как работает в Android**: ViewModel survives configuration changes, View наблюдает за LiveData/StateFlow — обновления автоматические. Data Binding или Compose упрощают привязку.
- **Плюсы**: автоматические обновления UI, lifecycle-aware (LiveData/Flow), ViewModel легко тестировать, минимум boilerplate.
- **Минусы**: сложнее отлаживать реактивные потоки, возможен over-use observable.

Strong/weak/soft/phantom reference

В Java (и соответственно в Kotlin) существуют четыре основных типа ссылок на объекты в куче: strong, weak, soft и phantom.
Они определяют, как Garbage Collector (GC) взаимодействует с объектом и когда он может его собрать.
По умолчанию все обычные ссылки — strong.
Специальные типы ссылок находятся в пакете java.lang.ref и используются для тонкого контроля над памятью (кэши, слушатели, cleanup).

Strong Reference (обычная сильная ссылка)

Это стандартная ссылка, которую мы используем повседневно: Object obj = new Object();.
Пока существует хотя бы одна strong-ссылка на объект, GC **никогда** не соберёт его, даже если памяти мало.
Объект становится доступным для сборки только после того, как все strong-ссылки на него исчезнут (становятся null или переприсваиваются).
Это основной механизм, обеспечивающий нормальную работу программы.
Проблема: может привести к memory leak, если забыть очистить ссылку (например, в статических коллекциях).

Weak Reference (слабая ссылка)

Создаётся через WeakReference<T> ref = new WeakReference<>(object);.
GC **может собрать** объект сразу же, как только на него не останется strong-ссылок, независимо от объёма памяти.
Получить объект: T obj = ref.get(); — может вернуть null, если уже собран.
Используется для кэшей, где данные можно пересоздать (например, WeakHashMap), и для избежания утечек (слушатели событий).
В Android: часто для Context в нестатических внутренних классах или в библиотеках вроде LeakCanary.

Soft Reference (мягкая ссылка)

Создаётся через SoftReference<T> ref = new SoftReference<>(object);.
GC собирает объект **только при нехватке памяти** (перед OutOfMemoryError).
Идеально для кэшей, где данные дорого пересоздавать (изображения, большие вычисления).
Объект живёт дольше weak, но всё равно может быть собран.
В Android: раньше использовали для bitmap-кэша, сейчас чаще LruCache.

Phantom Reference (фантомная ссылка)

Создаётся через PhantomReference<T> ref = new PhantomReference<>(object, referenceQueue);.
Это самая слабая ссылка: ref.get() всегда возвращает null — нельзя получить сам объект.
Объект считается достижимым, пока не очищен, но служит только для уведомления о сборке.
После сборки объекта ссылка помещается в ReferenceQueue, откуда можно получить уведомление.
Используется для выполнения cleanup-действий после финализации (например, освобождение нативных ресурсов).

Как lifecycle влияет на ресурсы

Очистка ресурсов в Android обычно происходит в методах lifecycle, таких как onDestroy() в Activity или Fragment, где закрываются соединения (например, базы данных, подписки на LiveData, сетевые клиенты) или освобождаются тяжёлые объекты (Bitmap, Cursor). Это предотвращает memory leaks и обеспечивает эффективное использование памяти при пересоздании компонентов.

REST

REST — это архитектурный стиль для проектирования сетевых приложений, прежде всего веб-API. Он был предложен Роем Филдингом в 2000 году в его диссертации и стал де-факто стандартом для современных веб-сервисов.
Основные принципы REST (6 ограничений):

Client-Server

Клиент и сервер разделены: клиент отвечает за UI, сервер — за логику и данные. Это позволяет независимо развивать обе части.

Stateless

Каждый запрос от клиента должен содержать всю необходимую информацию для его обработки. Сервер не хранит состояние сессии между запросами. Это упрощает масштабирование и повышает надёжность.

Cacheable

Ответы сервера должны явно указывать, можно ли их кэшировать (через заголовки Cache-Control, ETag и т.д.). Это снижает нагрузку на сервер и ускоряет работу клиента.

Uniform Interface (единообразный интерфейс)

Ключевой принцип, включающий:
Идентификация ресурсов через URI (например, /users/123).
Манипуляция ресурсами через стандартные HTTP-методы.
Самоописательные сообщения (заголовки + тело).
HATEOAS (Hypermedia as the Engine of Application State) — ответы содержат ссылки на связанные ресурсы (редко используется полностью).

Layered System

Архитектура может состоять из слоёв (прокси, балансировщики, кэши), клиент не знает, с каким именно слоем общается.

Code on Demand (опционально)

Сервер может отправлять исполняемый код клиенту (например, JavaScript), расширяя функциональность клиента.

Как работает REST на практике

Ресурсы: всё моделируется как ресурсы (users, orders, products).

HTTP-методы:
GET — чтение (безопасный, идемпотентный).
POST — создание.
PUT/PATCH — обновление (PUT идемпотентный).
DELETE — удаление.

Статусы HTTP: 200 OK, 201 Created, 404 Not Found, 500 Internal Server Error и т.д.

Форматы данных: чаще JSON, иногда XML.

Преимущества REST

- Простота и понятность.
- Масштабируемость (stateless).
- Кэшируемость.
- Совместимость с веб-инфраструктурой (браузеры, прокси).

Недостатки

- Не всегда подходит для сложных операций (нужны несколько запросов).
- Over-fetching/under-fetching (получаешь лишние или недостающие данные).
- Альтернативы: GraphQL, gRPC.

В Android-разработке REST — основной способ общения с бэкендом (Retrofit, OkHttp).
Большинство публичных API (Google, GitHub, Twitter) — RESTful.

Integration test
Тест взаимодействия компонентов (API + DB + UI), проверяет интеграцию, использует mocks или реальные сервисы.

TDD, BDD
TDD: тесты перед кодом, цикл red-green-refactor.
BDD: фокус на поведении, сценарии Given-When-Then, инструменты Cucumber.

Pure function
Функция без side effects, одинаковый ввод — одинаковый вывод, не меняет состояние.

Side effect
Изменение внешнего состояния: мутация, IO, логи.

ООП помогает в тестировании Android-приложений, предоставляя чёткие границы (инкапсуляция), интерфейсы для моков (например, мок Repository в ViewModel-тестах) и полиморфизм для инъекции зависимостей, что позволяет изолированно тестировать логику без реальных Android-компонентов.

Алгоритмы и Структуры Данных

Что такое временная и пространственная сложность алгоритма?

Временная сложность алгоритма описывает, как количество операций растёт с увеличением размера входных данных, обычно выражается в терминах количества элементарных шагов. Пространственная сложность оценивает объём дополнительной памяти, необходимой алгоритму (не включая входные данные), включая стек рекурсии или вспомогательные структуры. Эти метрики помогают прогнозировать производительность и масштабируемость кода, особенно в мобильных приложениях, где ресурсы ограничены.

Как обозначается Big O, Omega и Theta нотация?

Big O (O(f(n))) обозначает верхнюю границу роста функции — худший случай или асимптотическую верхнюю оценку. Omega (Ω(f(n))) — нижнюю границу, лучший случай или асимптотическую нижнюю оценку. Theta (Θ(f(n))) — точную границу, когда верхняя и нижняя оценки совпадают, то есть алгоритм ведёт себя именно как f(n) в асимптотике.

Примеры алгоритмов с сложностью O(1), O(log n), O(n), O(n log n), O(n^2)?

O(1): доступ по индексу в массиве или HashMap.get (constant time). O(log n): бинарный поиск в отсортированном массиве. O(n): линейный поиск или проход по списку. O(n log n): эффективные сортировки вроде QuickSort или MergeSort в среднем случае. O(n^2): вложенные циклы, как Bubble Sort или наивный поиск подстроки.

Что такое массив и его преимущества/недостатки? Как работает доступ по индексу в массиве.

Массив — непрерывный блок памяти фиксированного размера для хранения элементов одного типа. Преимущества: быстрый доступ по индексу O(1), эффективное использование кэша. Недостатки: фиксированный размер, дорогая вставка/удаление (сдвиг элементов). Доступ по индексу работает через вычисление адреса: base_address + index * element_size, что позволяет мгновенный переход.

Что такое динамический массив (ArrayList в Java/Kotlin)? Как ArrayList расширяется при добавлении элементов.

ArrayList — динамический массив, реализующий интерфейс List, с автоматическим изменением размера. Под капотом использует обычный массив, который при переполнении (load factor) копируется в новый больший (обычно в 1.5 раза). Это обеспечивает амортизированную O(1) вставку в конец, но иногда O(n) при resiz'e.

Пример использования SparseArray/SparseIntArray в Android и почему они эффективнее HashMap для разреженных данных?

SparseArray/SparseIntArray — специализированные структуры Android для хранения пар key-value, где key — int. Пример: хранение View по ID в Activity. Они эффективнее HashMap для разреженных данных (много пропусков в ключах), потому что используют два массива (keys и values) с бинарным поиском вместо хэширования и боксинга Integer, экономя память и время.

Основные операции со строками и их сложность. Как работает String Pool в Java?

Основные операции: конкатенация O(n), substring O(1) в Java 7+, length O(1), charAt O(1). String immutable, поэтому конкатенация создаёт новые объекты. String Pool — область heap для хранения строковых литералов; одинаковые литералы ссылаются на один объект, экономя память (intern() добавляет в пул).

Что такое односвязный и двусвязный список? Преимущества LinkedList над ArrayList.

Односвязный список — каждый узел содержит данные и ссылку на следующий. Двусвязный — плюс ссылка на предыдущий. Преимущества LinkedList: O(1) вставка/удаление в известной позиции (без сдвига), динамический размер. Недостатки: O(n) доступ по индексу, больше памяти на ссылки.

Когда в Android лучше использовать LinkedList? Основные операции (add, remove, get) и их сложность в связанном списке.

LinkedList лучше в Android для частых вставок/удалений в середине (например, очередь задач). Операции: add в конец O(1), remove известного узла O(1), get по индексу O(n) (проход по ссылкам).

Что такое стек (Stack) и очередь (Queue)? Примеры использования очереди в Android (Handler, MessageQueue). Что такое Deque и PriorityQueue?

Стек — LIFO структура (push/pop). Очередь — FIFO (enqueue/dequeue). В Android очередь используется в Handler/MessageQueue для обработки сообщений в главном потоке. Deque — двусторонняя очередь (добавление/удаление с обоих концов). PriorityQueue — очередь с приоритетом (heap-based).

Что такое хэш-таблица (HashMap/HashSet)? Как работает хэширование и разрешение коллизий (chaining, open addressing)? Сложность операций в HashMap (average и worst case).

HashMap — структура key-value на основе хэша ключа для быстрого доступа. Хэширование: hashCode() → индекс в массиве. Коллизии: chaining (списки в бакетах) или open addressing (пробинг). Сложность average O(1), worst O(n) при всех коллизиях.

Что такое load factor и rehashing?

Load factor — отношение заполненных бакетов к размеру массива (обычно 0.75). Rehashing — при превышении load factor массив увеличивается, все элементы перехэшируются в новый массив для сохранения производительности.

Разница HashMap и LinkedHashMap?

LinkedHashMap сохраняет порядок вставки (или доступа) через дополнительный двусвязный список, в отличие от обычного HashMap с произвольным порядком.

Когда использовать HashMap в Android, а когда SparseArray?

HashMap для любых ключей (объекты), SparseArray для int-ключей — экономит память (без боксинга Integer) и быстрее для разреженных данных.

Что такое бинарное дерево? Пример дерева в Android (например, View hierarchy).

Бинарное дерево — структура, где каждый узел имеет до двух детей. В Android View hierarchy — дерево ViewGroup с дочерними View, используемое для отрисовки и событий.

Что такое граф? Ориентированный и неориентированный? Способы представления графа (матрица смежности, список смежности).

Граф — набор вершин и рёбер. Неориентированный — рёбра без направления, ориентированный — с направлением. Представление: матрица смежности (2D массив) для плотных графов, список смежности (массив списков) для разреженных — экономит память.

Что такое DFS и BFS? Разница DFS и BFS по памяти и применению. Когда использовать BFS в Android (например, поиск кратчайшего пути в навигации)?

DFS (Depth-First Search) — углубление по ветви, использует стек. BFS (Breadth-First Search) — по уровням, использует очередь. DFS экономит память (O(h)), BFS — O(w). BFS для кратчайшего пути в невзвешенном графе, в Android — навигация или поиск соседних элементов.

Что такое цикл в графе и как его обнаружить?

Цикл — путь, возвращающийся в начальную вершину. Обнаружение: DFS с visited и parent, или Union-Find для неориентированных.

Сортировки:

Bubble Sort — многократный проход по массиву с обменом соседних элементов, если они в неправильном порядке, сложность O(n²).

Insertion Sort — вставка элементов в отсортированную часть массива по одному, эффективен для маленьких данных.

Selection Sort — поиск минимума в неотсортированной части и обмен с началом, O(n²).

Merge Sort — divide-and-conquer: делит массив пополам, сортирует рекурсивно и сливает, стабильная O(n log n).

Quick Sort — divide-and-conquer: выбор pivot, разделение массива и рекурсия, средняя O(n log n), быстрее Merge на практике из-за кэш-локальности.

Стабильные сортировки сохраняют порядок равных элементов (MergeSort), нестабильные — нет (QuickSort).

Collections.sort в Android использует TimSort (гибрид Merge + Insertion), стабильный O(n log n).

Кэш-локальность (cache locality) в контексте QuickSort относится к эффективному использованию процессорного кэша благодаря особенностям доступа к памяти во время выполнения алгоритма.

QuickSort работает «in-place» — сортирует массив без создания дополнительных больших структур данных, выполняя обмены элементов непосредственно в исходном массиве. Когда алгоритм выбирает pivot и разделяет массив на части (partitioning), он последовательно проходит по соседним элементам, сравнивая и меняя их местами. Такой последовательный доступ к памяти (sequential memory access) хорошо соответствует организации кэша процессора: при загрузке одного элемента в кэш-линию автоматически подгружаются соседние элементы, что минимизирует дорогостоящие обращения к оперативной памяти.

В результате QuickSort демонстрирует высокую кэш-локальность, особенно по сравнению с MergeSort, который требует дополнительного массива для слияния и выполняет множество разрозненных обращений к памяти при копировании данных. Это одна из главных причин, почему QuickSort на практике часто оказывается быстрее MergeSort при одинаковой асимптотической сложности O(n log n) — лучшее использование кэша снижает реальное время выполнения на современных процессорах.

Поиски:

Линейный поиск — последовательный перебор элементов, сложность O(n).

Бинарный поиск — деление отсортированного массива пополам, требует отсортированных данных, сложность O(log n).

Пример бинарного поиска в Android — поиск в отсортированном списке или Arrays.binarySearch.

Когда рекурсия предпочтительнее итерации?

Рекурсия предпочтительнее для задач с естественной рекуррентной структурой (деревья, графы, divide-and-conquer), когда код проще и читабельнее, но итерация лучше для производительности и избежания stack overflow в глубоких случаях.

Что такое динамическое программирование?

Динамическое программирование — метод оптимизации, разбивающий задачу на подзадачи, сохраняющий результаты для повторного использования, чтобы избежать экспоненциальной сложности.

Разница memoization и tabulation.

Memoization — top-down рекурсия с кэшем результатов подзадач. Tabulation — bottom-up итеративное заполнение таблицы от простых к сложным подзадачам.

Пример DP: вычисление Фибоначчи. Пример DP в Android (например, оптимизация RecyclerView DiffUtil).

Фибоначчи DP: храним уже вычисленные значения в массиве или map, снижая сложность с экспоненциальной до O(n). В Android DiffUtil использует DP (алгоритм Майерса) для эффективного вычисления различий между списками в RecyclerView.

Алгоритм Майерса (Myers' algorithm) — это эффективный алгоритм вычисления минимальной разницы (difference) между двумя последовательностями, разработанный Юджином Майерсом в 1986 году. Он используется для нахождения кратчайшей последовательности редактирования (shortest edit script), то есть минимального набора операций вставки, удаления и замены, необходимых для преобразования одной последовательности в другую.

Основная идея алгоритма заключается в поиске самого длинного общего подпоследовательности (LCS — Longest Common Subsequence) с помощью динамического программирования, но с оптимизацией по памяти и времени: сложность O(ND), где N — сумма длин последовательностей, D — количество различий (в худшем случае $O(N^2)$, но на практике значительно лучше).

В Android-разработке алгоритм Майерса лежит в основе **DiffUtil** (из Jetpack), который эффективно вычисляет различия между старым и новым списками данных в RecyclerView. Это позволяет обновлять только изменённые элементы, а не перерисовывать весь список, что существенно повышает производительность и плавность анимаций при обновлении UI. Без такого алгоритма простое сравнение списков было бы слишком медленным для больших коллекций.

Что такое LruCache и как работает? Пример использования LruCache для кэша изображений.

LruCache — кэш с фиксированным размером, удаляющий least recently used элементы при переполнении (LinkedHashMap с access order). В Android используется для кэша изображений (Bitmap), экономя память и ускоряя загрузку.

Что такое BitSet и когда использовать?

BitSet — структура для хранения битов, эффективная для флагов или множеств больших индексов. Использовать для разреженных булевых массивов или фильтров.

Что такое Bloom Filter (концепция)?

Bloom Filter — вероятностная структура для проверки принадлежности элемента множеству с возможными ложными срабатываниями, но без ложных отрицаний, экономит память за счёт хэшей.

Как выбрать структуру данных для частых поисков по ID в Android?

Для частых поисков по int-ID — SparseArray (быстрее и меньше памяти). Для объектных ключей — HashMap. Для сортированных — TreeMap.

Как алгоритмы влияют на ANR в Android?

Плохие алгоритмы (O(n²) на большом n) блокируют главный поток, вызывая ANR (Application Not Responding) при задержке >5 секунд.

Почему O(n²) алгоритмы опасны в onDraw или RecyclerView?

В onDraw или bindViewHolder O(n²) на большом списке приводит к дропам фреймов, лагам UI или ANR, так как операции в главном потоке.

Как оптимизировать поиск в большом списке в Android?

Использовать HashMap для O(1) по ключу, бинарный поиск для отсортированного списка или индексы в БД.

Пример, где плохой выбор структуры данных приводит к OOM.

HashMap<Integer, Bitmap> для тысяч изображений — боксинг Integer + overhead → OOM; лучше SparseArray или LruCache.

Как алгоритмы помогают в обработке больших JSON в Android?

Gson/Moshi с streaming (JsonReader) позволяют парсить большие JSON поэлементно O(n), избегая загрузки всего в память и OOM.

Java Collections

Основные интерфейсы в Collections: Collection, List, Set, Map, Queue.

Java Collections Framework включает ключевые интерфейсы для работы с группами объектов. Collection — корневой интерфейс для всех коллекций (кроме Map). List — упорядоченная коллекция с дубликатами и доступом по индексу. Set — коллекция уникальных элементов без дубликатов. Map — отображение ключ-значение, где ключи уникальны. Queue — коллекция для хранения элементов в порядке обработки (FIFO), с поддержкой приоритетов.

Иерархия Collection: какие интерфейсы от чего наследуются?

Иерархия начинается с Iterable (для for-each). Collection наследует Iterable. List, Set и Queue наследуют Collection. SortedSet наследует Set, NavigableSet — SortedSet. Map отдельный, не наследует Collection. Deque наследует Queue. Это обеспечивает общий API для итерации и операций.

Разница между Collection и Collections?

Collection — интерфейс для коллекций (List, Set и т.д.). Collections — утилитарный класс с статическими методами (sort, synchronizedList, unmodifiableList и т.д.). Collection определяет поведение, Collections предоставляет вспомогательные функции.

Что такое Iterable и Iterator?

Iterable — интерфейс, позволяющий объекту быть целью for-each, с методом iterator(). Iterator — объект для последовательного прохода по коллекции (hasNext, next, remove). Это базовый механизм итерации в Collections.

Что такое fail-fast и fail-safe итераторы? Пример fail-fast поведения в Android.

Fail-fast итератор бросает ConcurrentModificationException при структурной модификации коллекции во время итерации. Fail-safe работает с копией или позволяет модификации (CopyOnWriteArrayList). В Android fail-fast в ArrayList: изменение списка в цикле for-each вызывает исключение.

Что такое List? Основные характеристики. Когда использовать ArrayList, а когда LinkedList?

List — упорядоченная коллекция с дубликатами и доступом по индексу. ArrayList — динамический массив, быстрый доступ O(1), медленная вставка в середину. LinkedList — двусвязный список, быстрая вставка/удаление O(1), медленный доступ O(n). ArrayList для чтения/поиска, LinkedList для частых вставок в середину.

Что такое Vector и почему он устарел?

Vector — синхронизированная версия ArrayList из старых версий Java. Устарел, потому что синхронизация на каждый метод снижает производительность, лучше использовать Collections.synchronizedList или Concurrent коллекции.

Что такое CopyOnWriteArrayList и когда использовать?

CopyOnWriteArrayList — потокобезопасный List, создающий копию массива при модификации. Итераторы не бросают исключение. Использовать для "читай много, пиши редко" (например, слушатели событий).

Что такое Set? Основные характеристики. Разница между HashSet, LinkedHashSet и TreeSet? Как HashSet обеспечивает уникальность элементов?

Set — коллекция уникальных элементов без дубликатов. HashSet — неупорядоченный, быстрый O(1). LinkedHashSet — порядок вставки. TreeSet — сортированный по Comparable/Comparator. Уникальность в HashSet через hashCode и equals.

Как работает hashCode и equals в HashSet?

HashSet использует hashCode для бакета, equals для проверки коллизий. Правильная реализация обеспечивает корректную уникальность.

Почему LinkedHashSet сохраняет порядок вставки?

LinkedHashSet использует HashMap + двусвязный список для сохранения порядка вставки при итерации.

Когда использовать TreeSet вместо HashSet?

TreeSet для сортированных данных (natural order или Comparator), когда нужен порядок или навигация (floor, ceiling).

Что такое EnumSet и BitSet?

EnumSet — специализированный Set для enum, эффективный (битовый). BitSet — массив битов для флагов или больших индексов.

Что такое Map? Основные характеристики. Разница между HashMap, LinkedHashMap и TreeMap?
Map — это интерфейс, представляющий коллекцию пар «ключ—значение», где каждый ключ уникален и сопоставлен с одним значением. Основные характеристики: ключи не могут повторяться, значения могут; доступ, добавление и удаление по ключу. HashMap — неупорядоченная реализация, обеспечивает среднюю сложность O(1) для основных операций, но порядок элементов не гарантируется. LinkedHashMap сохраняет порядок вставки (или порядок доступа, если включён режим access-order), что полезно, когда нужно итерировать элементы в том порядке, в котором они были добавлены. TreeMap сортирует ключи по естественному порядку (Comparable) или заданному компаратору, сложность операций O(log n), но позволяет выполнять навигацию (floor, ceiling, subMap).

Что изменилось в HashMap в Java 8 (деревья в бакетах)?
В Java 8 в HashMap была добавлена оптимизация для случаев с большим количеством коллизий: если в одном бакете (ячейке массива) накапливается более 8 элементов, связанный список преобразуется в красно-чёрное дерево. Это снижает worst-case сложность поиска, вставки и удаления с O(n) до O(log n) при большом числе коллизий (например, при плохом hashCode). При уменьшении количества элементов дерево может обратно превратиться в список. Это улучшило производительность в экстремальных сценариях, не затрагивая средний случай O(1).

Что такое IdentityHashMap?
IdentityHashMap — это специальная реализация Map, которая использует для сравнения ключей оператор == (проверку на идентичность объектов), а не метод equals(). Это означает, что два разных объекта, даже если equals() возвращает true, будут считаться разными ключами. Используется в ситуациях, когда нужно различать объекты по ссылке, например, при работе с экземплярами классов, где equals() переопределён, но требуется именно уникальность по памяти.

Что такое WeakHashMap и когда использовать?
WeakHashMap — это Map, в котором ключи хранятся через WeakReference. Это означает, что если на ключ больше нет сильных ссылок, он (и соответствующая пара ключ-значение) может быть удалён сборщиком мусора в любой момент. WeakHashMap полезен для реализации кэшей, где ключ — это объект, который может больше не использоваться в программе (например, кэш метаданных объектов), и мы хотим, чтобы записи автоматически очищались при GC, не удерживая объекты в памяти.

Что такое ConcurrentHashMap и его преимущества? Разница HashMap и ConcurrentHashMap?

ConcurrentHashMap — потокобезопасная реализация Map, позволяющая множественным потокам одновременно читать и записывать данные без блокировки всего объекта. Преимущества: высокая конкурентность, хорошая масштабируемость при большом числе потоков, lock-free чтение. HashMap не потокобезопасен — при одновременной модификации может привести к повреждению структуры. ConcurrentHashMap работает быстрее в многопоточной среде, чем Collections.synchronizedMap(HashMap), который блокирует весь объект.

Как ConcurrentHashMap обеспечивает потокобезопасность?

ConcurrentHashMap делит внутренний массив на сегменты (в Java 8+ — до 16 сегментов, в новых версиях — динамически). При записи поток блокирует только нужный сегмент, а не всю карту. Чтение выполняется без блокировок благодаря volatile-полям и CAS-операциям (Compare-And-Swap). В Java 8+ при коллизиях используются деревья, как в обычном HashMap, но с дополнительной синхронизацией на уровне узлов.

Что такое Queue и Deque? Разница между Queue и Deque?

Queue — интерфейс для очереди FIFO (First In — First Out), предоставляет методы для добавления в конец (offer/add) и извлечения из начала (poll/remove). Deque (Double-ended queue) расширяет Queue и позволяет добавлять/удалять элементы с обоих концов (addFirst/addLast, pollFirst/pollLast). Таким образом, Deque — более универсальная структура, которая может использоваться и как очередь, и как стек.

Что такое PriorityQueue и как работает?

PriorityQueue — очередь с приоритетом, которая извлекает элементы в порядке их естественного сравнения (Comparable) или по заданному Comparator. Под капотом реализована на основе двоичной куче (binary heap), что обеспечивает O(log n) для вставки и извлечения, O(1) для получения минимального/максимального элемента. Не гарантирует стабильность порядка равных элементов.

Что такое ArrayDeque и когда использовать?
ArrayDeque — эффективная реализация Deque на основе циклического массива. Поддерживает O(1) операции с обоих концов, не имеет overhead'а на узлы, как LinkedList. Используется как очередь или стек, когда нужна максимальная производительность и нет необходимости в случайном доступе по индексу. Рекомендуется как предпочтительная реализация Deque.

Разница PriorityQueue и PriorityBlockingQueue?
PriorityQueue — обычная очередь с приоритетом, не потокобезопасна. PriorityBlockingQueue — потокобезопасная версия, реализует BlockingQueue, блокирует поток при попытке извлечь из пустой очереди или добавить в полную (с заданной ёмкостью). Подходит для сценариев producer-consumer в многопоточной среде.

Что делает Collections.unmodifiableList/Set/Map? Что делает Collections.synchronizedList/Set/Map?
Collections.unmodifiableList/Set/Map создаёт неизменяемую (read-only) обёртку над коллекцией: все методы модификации (add, remove и т.д.) бросают UnsupportedOperationException. Collections.synchronizedList/Set/Map создаёт потокобезопасную обёртку, где каждый метод коллекции обёрнут в synchronized-блок, что делает её безопасной для использования несколькими потоками.

Разница между synchronized и concurrent коллекциями?
Synchronized-коллекции (Collections.synchronized*) блокируют весь объект при каждой операции, что безопасно, но сильно снижает производительность при высокой конкуренции. Concurrent-коллекции (ConcurrentHashMap, CopyOnWriteArrayList и т.д.) используют сегментную блокировку, CAS или копирование, обеспечивая лучшую конкурентность и масштабируемость в многопоточных приложениях.

Что такое unmodifiable vs immutable коллекции (Java 9+)?
Unmodifiable-коллекции (Collections.unmodifiableList и т.д.) — это view: исходная коллекция остаётся изменяемой, но через unmodifiable-обёртку модификация невозможна. Immutable-коллекции (List.of(), Set.of(), Map.of() с Java 9+) — это настоящие неизменяемые копии: сами данные защищены, никто не может их изменить даже через оригинал.

Как получить synchronized Map/List/Set?
Используйте статические методы Collections: synchronizedMap(new HashMap<>()), synchronizedList(new ArrayList<>()), synchronizedSet(new HashSet<>()).

Что такое Iterator и ListIterator? Разница Iterator и ListIterator?
Iterator — базовый итератор для всех Collection: методы hasNext(), next(), remove(). ListIterator — расширение Iterator, специфичное для List: позволяет двигаться в обе стороны (hasPrevious(), previous()), добавлять элементы (add()), заменять текущий (set()).

Что такое Spliterator (Java 8)? Как Spliterator используется в parallel streams?
Spliterator — итератор, способный разделяться на части (trySplit()), что позволяет эффективно обрабатывать коллекции параллельно. В parallel streams Spliterator делит данные на подмножества, каждое из которых обрабатывается в отдельном потоке, что ускоряет операции map, filter, reduce.

Что такое ConcurrentSkipListMap/Set?
ConcurrentSkipListMap/Set — потокобезопасные реализации Map/Set с сортировкой (на основе skip list). Поддерживают O(log n) операции и параллельный доступ, подходят для случаев, когда нужна одновременно сортировка и потокобезопасность.

Почему mutable объекты опасны как ключи в HashMap?
Если ключ изменяется после добавления в HashMap, его hashCode() может измениться, и он окажется в неправильном бакете. В результате get() не найдёт элемент, хотя он физически присутствует в коллекции, что приводит к логическим ошибкам.

Как правильно переопределять hashCode и equals?
Метод equals() должен проверять логическое равенство объектов. hashCode() должен возвращать одинаковые значения для равных объектов (контракт: если a.equals(b), то a.hashCode() == b.hashCode()). Используйте Objects.equals() и Objects.hash() для удобного и безопасного переопределения.

Что происходит, если equals изменён для ключа в HashMap?
Если после вставки изменить ключ так, что equals() начнёт возвращать другое значение, объект останется в старом бакете. При поиске по новому значению HashMap не найдёт ключ, хотя он физически присутствует.

Как избежать ConcurrentModificationException?
Не модифицировать коллекцию во время итерации for-each или Iterator. Использовать итератор явно и вызывать remove() только через него. Для многопоточной модификации применять concurrent-коллекции (ConcurrentHashMap, CopyOnWriteArrayList) или блокировки.

Как настроить HashMap для лучшей производительности?
Заранее задавать initialCapacity и loadFactor (по умолчанию 0.75). initialCapacity = (ожидаемое количество элементов / loadFactor) + 1. Использовать immutable ключи с хорошим hashCode, чтобы минимизировать коллизии.

Почему String/Integer хороши как ключи в Map?
String и Integer — immutable, их hashCode() и equals() надёжны и хорошо распределены. Не меняются после создания, поэтому ключ не «потеряется» в HashMap. String кэширует литералы, Integer кэширует значения -128..127.

Какие коллекции чаще используются в Android-разработке?
ArrayList — для списков данных, HashMap — для словарей (ID → объект), HashSet — для уникальных элементов, SparseArray/SparseIntArray — для int-ключей, LruCache — для кэша, ConcurrentHashMap — в многопоточных сценариях.

Почему в Android есть SparseArray/SparseIntArray/SparseBooleanArray?
Эти классы оптимизированы для хранения данных с int-ключами. Они избегают боксинга Integer и используют два массива (ключи и значения) с бинарным поиском, что экономит память и снижает overhead по сравнению с HashMap<Integer, V>.

Разница SparseArray и HashMap в Android? Когда использовать SparseArray вместо HashMap?

SparseArray использует два массива + бинарный поиск, не боксит ключи, потребляет меньше памяти. HashMap — хэш-таблица с боксингом Integer. SparseArray предпочтительнее при int-ключах и разреженных данных (например, View по ID).

Что такое LruCache и как работает? Пример LruCache для кэша изображений.

LruCache — кэш с политикой Least Recently Used: хранит ограниченное число элементов, при превышении размера удаляет наименее недавно использованный. Реализован на базе LinkedHashMap с access-order. Пример: LruCache<String, Bitmap> для кэширования изображений по URL — Bitmap загружается один раз, затем берётся из кэша.

Как ArrayMap отличается от HashMap в Android? Когда использовать ArrayMap вместо HashMap?

ArrayMap — компактная реализация Map на двух массивах (ключи и значения) + хэш. Работает медленнее HashMap при большом числе элементов, но потребляет меньше памяти для маленьких коллекций (< ~1000 элементов). Использовать ArrayMap в Android для небольших словарей (например, в Bundle, View-параметры).

Deque — это интерфейс (Double Ended Queue), расширяющий Queue и позволяющий добавлять/удалять элементы с обоих концов (голова и хвост). **ArrayDeque** — эффективная реализация Deque на основе циклического массива, обеспечивающая O(1) для операций на концах, без overhead на узлы и с лучшей производительностью по сравнению с LinkedList как Deque.

Как коллекции влияют на память в Android?

Коллекции создают множество объектов, особенно при боксинге примитивов (Integer вместо int). HashMap имеет overhead на бакеты и записи. Большие коллекции → больше GC-пауз и риск OOM. Разреженные данные или примитивы → лучше SparseArray/ArrayMap.

Как избежать memory leak с коллекциями в Android?

Не храните Context, Activity, Fragment, View в статических коллекциях или долгоживущих объектах. Используйте WeakReference для ключей/значений, если нужно. Очищайте коллекции в onDestroy/onStop. Применяйте LruCache вместо обычных Map для кэша.

GIT

Что такое Git и чем он отличается от GitHub?
Git — это распределённая система контроля версий, разработанная для отслеживания изменений в исходном коде проектов, позволяющая нескольким разработчикам работать над одним кодом одновременно без потери данных. Она работает локально на компьютере пользователя, храня всю историю изменений в репозитории, и поддерживает операции такие как создание веток, слияние изменений и восстановление версий. GitHub, в свою очередь, — это веб-платформа для хостинга Git-репозиториев, предоставляющая дополнительные инструменты для совместной работы, такие как pull requests, issues, code review и интеграция с CI/CD. Основное отличие заключается в том, что Git — это инструмент для контроля версий, а GitHub — облачный сервис, использующий Git как основу, но добавляющий социальные и коллаборативные функции для удалённого хранения и сотрудничества.

Что такое репозиторий (repository)?
Репозиторий в Git — это директория, где хранится весь проект с историей изменений, включая файлы кода, конфигурации и метаданные. Он представляет собой базу данных, содержащую все коммиты, ветки, теги и ссылки, позволяющую отслеживать эволюцию проекта во времени. Локальный репозиторий находится на компьютере разработчика и может быть синхронизирован с удалённым репозиторием на серверах вроде GitHub. Репозиторий инициализируется командой git init, и в нём хранится скрытая папка .git, содержащая всю внутреннюю информацию. В контексте Android-разработки репозиторий часто содержит код приложения, ресурсы, Gradle-файлы и историю версий для совместной работы команды.

Что такое commit? Как его создать?
Commit в Git — это фиксация изменений в репозитории, представляющая собой снимок состояния проекта на определённый момент времени, с уникальным идентификатором (SHA-1 хэш), сообщением и метаданными (автор, дата). Он позволяет сохранять историю изменений и возвращаться к предыдущим версиям. Чтобы создать commit, сначала добавьте файлы в staging area с помощью git add <файл>, затем выполните git commit -m "Сообщение о изменениях", где сообщение описывает суть коммита. В Android-разработке commit часто фиксирует добавление новой фичи или исправление бага, обеспечивая traceability в проекте.

Что такое staging area (индекс)? Как добавить файлы в staging?
Staging area, или индекс, в Git — это промежуточная область между рабочей директорией и репозиторием, где собираются изменения перед их фиксацией в commit. Она позволяет выбрать, какие модификации включить в следующий commit, исключая ненужные файлы. Чтобы добавить файлы в staging, используйте команду git add <файл> для конкретного файла или git add . для всех изменённых файлов. В Android это полезно для поэтапного добавления изменений в Gradle скрипты или XML-файлы, чтобы commit был атомарным и осмысленным.

Чем отличается git commit -a от обычного commit?
Опция -a в git commit -a -m "message" автоматически добавляет все отслеживаемые изменённые файлы в staging area перед коммитом, пропуская шаг git add. Обычный commit требует предварительного добавления файлов в staging. Это удобно для быстрых коммитов, но не добавляет новые файлы (только tracked). В Android это ускоряет фиксацию мелких правок в коде.

Что такое HEAD в Git?
HEAD — это указатель на текущий commit или ветку, в которой находится рабочая директория. Он представляет "голову" текущей ветки и обновляется при коммитах или checkout. HEAD может быть detached (не привязан к ветке), что полезно для экспериментов. В Android HEAD помогает в навигации по истории, например, при git checkout HEAD~1 для просмотра предыдущего состояния проекта.

Что такое branch и зачем нужны ветки?
Branch — это параллельная линия разработки, указатель на commit, позволяющий работать над функциями независимо от основной ветки. Ветки нужны для изоляции изменений (feature branches, bugfix), экспериментов и параллельной работы команды. В Android ветки используются для разработки новых фич (branch "feature/new-ui"), без влияния на master/main.

Как создать новую ветку? Как переключиться на неё?
Новая ветка создаётся командой git branch <имя-ветки>, где имя — описательное (например, "feature/login"). Чтобы переключиться, используйте git checkout <имя-ветки> или git switch <имя-ветки> (в новых версиях Git). Альтернатива: git checkout -b <имя> создаёт и сразу переключает. В Android это позволяет работать над новым экраном без риска сломать основной код.

Как слить (merge) ветку в основную?
Чтобы слить ветку, переключитесь на основную (например, git checkout main), затем выполните git merge <имя-ветки>. Git автоматически создаст merge-commit, если нет конфликтов. Если конфликты — разрешите вручную в файлах и закоммитьте. В Android merge используется для интеграции фичи в main после code review.

Что такое merge conflict и как его разрешить?
Merge conflict возникает, когда в сливаемых ветках изменены одни и те же строки файлов. Git помечает конфликты в файлах (<<<, ===, >>>). Чтобы разрешить, откройте файл, отредактируйте нужную версию, удалите маркеры, добавьте в staging (git add) и завершите merge коммитом. В Android конфликты часто в XML или Gradle, используйте инструменты вроде IntelliJ для разрешения.

Что такое pull request (PR) и зачем он нужен?
Pull request — запрос на слияние изменений из одной ветки в другую (обычно в main), с возможностью code review, обсуждения и тестов. Зачем нужен: обеспечивает качество кода, коллаборацию, интеграцию CI/CD. На GitHub PR создаётся через UI, с описанием, скриншотами и тегами.

Как создать pull request на GitHub?
На GitHub перейдите в репозиторий, выберите ветку с изменениями, кликните "New pull request". Выберите базовую ветку (main), добавьте заголовок, описание, назначьте ревьюеров. Добавьте labels, milestones. После создания PR проходит code review, тесты, и merge.

Что такое fork репозитория?
Fork — копия репозитория на вашем аккаунте GitHub, позволяющая вносить изменения без влияния на оригинал. Используется для вклада в open-source: форкните, измените, создайте PR в оригинал. В Android fork полезен для экспериментов с библиотеками.

Как сделать fork и внести изменения через pull request?
На GitHub кликните "Fork" в репозитории. Клонируйте свой fork локально (git clone <url>). Создайте ветку, внесите изменения, закоммитьте и запушьте. Затем создайте PR из своей ветки в оригинальный репозиторий. В Android это стандарт для вклада в библиотеки вроде Retrofit.

Отличие merge и rebase
Merge создаёт новый коммит (merge commit), который объединяет две ветки, сохраняя полную историю изменений обеих веток в виде графа с двумя родителями. История остаётся нелинейной, но полностью отражает, когда и как ветки развивались параллельно.
Команда: git merge feature-branch (находясь на main).
Rebase переписывает историю текущей ветки, перенося её коммиты поверх целевой ветки (обычно main). В результате получается линейная история, как будто все изменения делались последовательно в одной ветке.
Команда: git rebase main (находясь на feature-branch).

Что такое remote repository?
Remote repository — это версия вашего проекта, размещённая на удалённом сервере (GitHub, GitLab, Bitbucket и т.д.), которая служит центральным хранилищем кода для совместной работы команды. Он позволяет синхронизировать локальные изменения с коллегами через команды push и pull, а также выступает основой для code review, pull request и CI/CD. В отличие от локального репозитория, remote хранит историю в облаке и обеспечивает доступность проекта для всех участников. Обычно основной remote называется origin и добавляется автоматически при клонировании или явно через git remote add origin <url>.

Чем отличается git pull от git fetch + git merge?
Команда git pull — это комбинация двух операций: git fetch (загружает изменения из remote без их применения) и git merge (сливает загруженные изменения в текущую ветку). Таким образом, git pull сразу обновляет локальную ветку, что удобно, но менее контролируемо. Вариант git fetch + git merge позволяет сначала посмотреть, что именно будет влито (например, через git log origin/main..main), разрешить возможные конфликты осознанно или даже отменить merge, если изменения нежелательны. Это более безопасный и прозрачный подход, особенно в командах с частыми изменениями в основной ветке.

Что такое git clone и что он делает?
git clone <url> — команда, которая полностью копирует удалённый репозиторий на локальный компьютер, создавая локальную копию со всей историей коммитов, ветками и тегами. При этом автоматически создаётся remote с именем origin, указывающий на исходный URL, и выполняется checkout основной ветки (обычно main или master). Клонирование скачивает не только файлы текущего состояния, но и всю метаданную Git, что позволяет работать автономно и синхронизироваться позже через push/pull. В Android-проектах это стандартный первый шаг для начала работы над кодом.

Что такое git push --force и почему он опасен?
Что такое git push --force-with-lease?
git push --force принудительно переписывает историю на удалённом репозитории, заменяя её локальной версией ветки, даже если удалённая ветка содержит коммиты, которых нет локально. Это опасно, потому что может удалить чужие коммиты, нарушить работу коллег, которые уже основывались на старой версии ветки, и привести к потере истории.
git push --force-with-lease — более безопасная альтернатива: она проверяет, что удалённая ветка не изменилась с момента последнего fetch. Если кто-то успел запушить изменения, push отклоняется, предотвращая случайное удаление чужой работы. Это рекомендуемый способ в командах для безопасного переписывания истории.

Что такое protected branch на GitHub?
Protected branch — это ветка на GitHub (обычно main или master), для которой включены правила защиты: запрет прямого push'а, обязательный code review, проверка статуса CI/CD, ограничение на слияние только через pull request. Это предотвращает случайные или нерецензированные изменения в основной ветке, обеспечивает стабильность и качество кода. Настройки защиты находятся в разделе Settings → Branches репозитория.

Что такое GitHub Actions?
Как создать простой workflow (CI/CD) в Git Actions?
GitHub Actions — встроенная платформа CI/CD, позволяющая автоматизировать сборку, тестирование, деплой и другие задачи прямо в репозитории через YAML-файлы в директории .github/workflows. Простой workflow создаётся следующим образом: создайте файл .github/workflows/ci.yml, укажите on: [push, pull_request], затем добавьте job с runs-on: ubuntu-latest, шагами checkout, setup-java (для Android), gradlew build/test. Пример: запуск ./gradlew build и ./gradlew test на каждый push в main. Workflow запускается автоматически и отображается в разделе Actions репозитория.

Что такое GitHub Pages?
GitHub Pages — бесплатный сервис статического хостинга, позволяющий публиковать веб-сайты прямо из репозитория GitHub. Поддерживает HTML, CSS, JavaScript и генераторы сайтов (Jekyll, Hugo и т.д.). Активируется в настройках репозитория (Settings → Pages), можно выбрать ветку (обычно gh-pages или main) и папку (например, /docs). Используется для документации проектов, портфолио или демонстрации Android-приложений (например, статические скриншоты или демо-страницы).

Как работает GitHub Issues и Projects?
GitHub Issues — система для отслеживания задач, багов, улучшений и обсуждений в проекте. Каждая issue имеет заголовок, описание, labels, assignee, milestone и комментарии. GitHub Projects — Kanban-доска (или таблица), связанная с репозиторием, где issues и PR визуализируются как карточки (To Do → In Progress → Done). Поддерживает автоматизацию (например, перемещение issue в Done при merge PR). В Android-проектах это стандартный инструмент планирования фич и багфиксов.

Что такое GitHub Releases?
GitHub Releases — функциональность для публикации готовых версий проекта (APK, JAR, бинарники и т.д.) с тегами, changelog, описанием и прикреплёнными файлами. Releases привязаны к git-тегам и отображаются в разделе Releases репозитория. Поддерживают pre-releases и assets (например, APK для Android).

Как создать release с тегом и changelog?
Создайте тег локально (git tag v1.0.0 и git push origin v1.0.0) или через интерфейс GitHub. Перейдите в раздел Releases → Draft a new release → выберите тег (или создайте новый) → добавьте заголовок, описание (можно сгенерировать changelog из коммитов через «Generate release notes»), прикрепите файлы (например, app-release.apk) → отметьте «This is a pre-release» при необходимости → нажмите Publish release. В Android это стандартный способ публикации версий приложения.

Как отменить последний commit (git reset, git revert)?
Отмена последнего коммита может выполняться двумя основными способами в зависимости от того, был ли коммит уже запушен в удалённый репозиторий. Команда git reset --hard HEAD~1 полностью удаляет последний коммит и все изменения в рабочей директории, возвращая HEAD на предыдущий коммит (опасно для запушенных изменений, так как переписывает историю). Альтернатива — git revert HEAD, которая создаёт новый коммит, отменяющий изменения предыдущего, сохраняя историю неизменной и безопасно работая с уже опубликованными коммитами. В Android-проектах git revert предпочтительнее для main-ветки, чтобы не нарушить работу команды, а reset используется локально для быстрой очистки неудачных экспериментов.

Что такое cherry-pick и когда он нужен?
Cherry-pick — это команда git cherry-pick <commit-hash>, которая позволяет применить изменения из конкретного коммита другой ветки в текущую ветку, без слияния всей ветки. Это полезно, когда нужно перенести только одно исправление бага или небольшую фичу, не затрагивая остальную историю разработки. Например, если баг был исправлен в feature-ветке, но релиз уже вышел, cherry-pick позволяет быстро перенести фикс в release-ветку. Команда создаёт новый коммит с теми же изменениями, но новым хэшем, что требует осторожности при повторном применении (возможны конфликты).

Что такое stash и как его использовать?
Stash — это временное сохранение незакоммиченных изменений рабочей директории и индекса, чтобы быстро переключиться на другую ветку или задачу без создания коммита. Команда git stash сохраняет изменения в стек (последний сохранённый — сверху), git stash list показывает список, git stash apply или git stash pop применяет их обратно (pop удаляет из стека). В Android это удобно при срочном переключении на багфикс, когда в рабочей ветке остались недоделанные изменения — их можно «спрятать», пофиксить баг и вернуть stash.

Как работать с submodule в Git?
Submodule — это способ включить другой репозиторий как поддиректорию в основной проект, сохраняя независимую историю. Добавляется командой git submodule add <url> path, после чего в репозитории появляется файл .gitmodules и ссылка на коммит подмодуля. При клонировании проекта нужно выполнить git submodule update --init --recursive, чтобы загрузить содержимое. Обновление: git submodule update --remote или переход в подмодуль и pull. В Android submodules часто используются для библиотек (например, Room, Retrofit), чтобы фиксировать конкретную версию зависимости.

Что такое .gitignore и как правильно его настроить?
Файл .gitignore — это текстовый файл в корне репозитория, содержащий шаблоны файлов и директорий, которые Git должен игнорировать при коммитах (например, build/, *.apk, local.properties). Он предотвращает попадание временных файлов, кэшей Gradle, ключей API и локальных настроек в репозиторий. Правильная настройка включает использование готовых шаблонов (например, от github/gitignore для Android), игнорирование больших файлов (через Git LFS для бинарников) и проверку через git status --ignored. В Android типичный .gitignore исключает build/, .idea/, *.iml, local.properties и keystore-файлы.

Как исправить последний commit (git commit --amend)?
Команда git commit --amend позволяет изменить сообщение последнего коммита или добавить в него пропущенные файлы без создания нового коммита. Если нужно добавить изменения, сначала git add нужные файлы, затем git commit --amend (или --amend -m "новое сообщение"). Если коммит уже запушен, потребуется git push --force-with-lease (или --force), но это опасно в команде. В Android это удобно для исправления мелких опечаток в сообщении коммита или добавления забытого ресурса.

Как удалить файл из истории Git (git filter-branch / BFG Repo-Cleaner)?
Удаление файла из всей истории Git требуется, когда в репозиторий попали чувствительные данные (ключи, пароли). Классический способ — git filter-branch --force --index-filter 'git rm --cached --ignore-unmatch путь/к/файлу' --prune-empty --tag-name-filter cat -- --all, после чего нужно принудительно запушить изменения. Более современный и быстрый инструмент — BFG Repo-Cleaner (Java-утилита), который проще в использовании и быстрее работает с большими репозиториями. После очистки рекомендуется изменить все пароли/ключи.

Что такое GitHub CLI и как его использовать?
GitHub CLI (gh) — официальный командный интерфейс GitHub для работы с репозиториями прямо из терминала. Устанавливается через brew, scoop или пакетный менеджер. Основные команды: gh repo create, gh pr create, gh issue list, gh release create. Например, gh pr create --title "Fix bug" --body "Описание" создаёт pull request. В Android это ускоряет работу с PR и релизами без перехода в браузер.

Как настроить SSH-ключи для GitHub?
Для настройки SSH-ключей сгенерируйте пару ключей командой ssh-keygen -t ed25519 -C "your@email.com", сохраните в ~/.ssh/id_ed25519. Скопируйте публичный ключ (cat ~/.ssh/id_ed25519.pub), добавьте его в GitHub → Settings → SSH and GPG keys → New SSH key. Проверьте подключение командой ssh -T git@github.com. После этого git clone/push будет работать без ввода логина/пароля.

Что такое GitHub token (PAT) и как его использовать?
Personal Access Token (PAT) — это токен аутентификации вместо пароля для операций через HTTPS. Создаётся в GitHub → Settings → Developer settings → Personal access tokens → Tokens (classic) → Generate new token (выберите scopes: repo, workflow и т.д.). Токен используется вместо пароля: git clone https://<username>:<token>@github.com/user/repo.git или в credential helper. PAT необходим для GitHub Actions, скриптов и CI.

Как настроить GitHub Actions для Android-проекта (build, tests, lint)?
Для Android-проекта создайте файл .github/workflows/android.yml. Пример: триггер on: [push, pull_request], job на runs-on: ubuntu-latest с шагами checkout, установка JDK, Gradle cache, ./gradlew build test lint. Добавьте secrets для подписывания APK. Workflow может включать upload-artifact для APK и уведомления в Slack/Telegram

Что такое Dependabot и как он работает?
Dependabot — встроенный инструмент GitHub для автоматического обновления зависимостей (Gradle, npm, Maven и т.д.). Он сканирует репозиторий, находит устаревшие версии библиотек, создаёт pull request с обновлением и changelog. Настраивается в .github/dependabot.yml (указание экосистемы, интервал проверки). В Android Dependabot обновляет версии в build.gradle и создаёт PR с комментарием о безопасности и изменениях.

Multithreading

Что такое процесс и поток? В чём разница?
Процесс — это независимая единица выполнения операционной системы, которая имеет собственное адресное пространство, ресурсы (файлы, память, дескрипторы) и изолирована от других процессов. Каждый процесс запускается операционной системой и может содержать один или несколько потоков. Поток (thread) — это наименьшая единица выполнения внутри процесса, которая делит с другими потоками того же процесса адресное пространство, открытые файлы и ресурсы, но имеет собственный стек вызовов, регистры процессора и счётчик команд. Основное различие заключается в изоляции: процессы полностью изолированы друг от друга (переключение между ними дорогое), тогда как потоки внутри одного процесса работают в общей памяти и переключаются гораздо быстрее. В Android каждый процесс имеет свой собственный Dalvik/ART VM, а приложение обычно запускается в одном процессе, внутри которого и выполняются все потоки.

Что такое Main Thread (UI Thread) в Android и почему он критически важен?
Main Thread (он же UI Thread) — это основной поток Android-приложения, созданный системой при запуске процесса приложения. Именно в нём обрабатываются все события пользовательского интерфейса: отрисовка экрана, обработка касаний, нажатий, анимации, обновление View и вызовы методов жизненного цикла Activity/Fragment. Этот поток критически важен, потому что Android строго запрещает выполнять длительные операции в нём: любая блокировка дольше нескольких секунд приводит к появлению диалога ANR (Application Not Responding), приложение считается «зависшим», и пользователь может его принудительно закрыть. Кроме того, все изменения UI-элементов (вызовы setText, setVisibility и т.д.) должны происходить именно в Main Thread — иначе возникает исключение CalledFromWrongThreadException. Именно поэтому все тяжёлые операции (сеть, база данных, обработка файлов) выносятся в фоновые потоки или корутины.

Что такое асинхронность и синхронность в контексте Android-приложения?
Синхронность означает, что операция выполняется последовательно: текущий поток блокируется до полного завершения задачи (например, сетевой запрос на Main Thread — приложение «замерзает» до получения ответа). Асинхронность позволяет запускать длительную операцию в фоновом потоке или корутине, не блокируя основной поток: задача выполняется параллельно, а результат обрабатывается позже через callback, LiveData/StateFlow или suspend-функцию. В Android асинхронность критически важна для поддержания отзывчивости интерфейса: пользователь может продолжать взаимодействовать с приложением, пока идёт загрузка данных, парсинг JSON или запись в базу. Примеры асинхронных механизмов: Coroutines, WorkManager, Retrofit с suspend, Room с Flow/StateFlow.

Можно ли выполнять сетевые запросы и работу с БД на Main Thread?
Нет, выполнять сетевые запросы и операции с базой данных на Main Thread категорически запрещено в Android начиная с API 11 (Android 3.0). Система намеренно выбрасывает NetworkOnMainThreadException при попытке сетевого запроса и блокирует UI при длительных операциях с БД, потому что это приводит к зависанию интерфейса и ANR. Сетевые запросы могут занимать от сотен миллисекунд до десятков секунд (в зависимости от сети), а операции с БД — от миллисекунд до секунд при сложных запросах. Это нарушает принцип отзывчивости: пользователь видит «замерзший» экран и не может взаимодействовать с приложением. Современные библиотеки (Retrofit с suspend, Room с Flow, WorkManager) изначально проектируются для асинхронной работы.

Как создать и запустить поток в Kotlin/Java?
В Java/Kotlin поток создаётся несколькими способами: через наследование от класса Thread (class MyThread : Thread() { override fun run() { ... } }), через реализацию интерфейса Runnable (val runnable = Runnable { ... }; Thread(runnable).start()), или через лямбду (Thread { ... }.start()). В Kotlin чаще используют корутины, но классический способ остаётся актуальным для низкоуровневых задач. Запуск потока выполняется исключительно методом start(), который создаёт новый поток ОС и вызывает метод run(). Прямой вызов run() просто выполнит код в текущем потоке, без создания нового. В Android создание потоков вручную используется редко — предпочтительнее Coroutines или WorkManager.

Разница между Thread.start() и Thread.run()?
Метод start() запускает новый поток операционной системы, создаёт стек вызовов и вызывает метод run() в этом новом потоке. После вызова start() поток переходит в состояние runnable и начинает независимое выполнение. Прямой вызов run() просто выполняет код метода в текущем потоке, без создания нового — это обычный вызов функции. Ошибка новичков — вызов run() вместо start(), из-за чего код выполняется синхронно в вызывающем потоке, а не асинхронно. В Android это критично: если вызвать run() на потоке, который должен обновлять UI, изменения могут произойти не в Main Thread.

Что такое Daemon Thread?
Daemon Thread (демон-поток) — это фоновый поток, который автоматически завершается, когда все не-демон потоки (user threads) в приложении заканчивают работу. JVM не ждёт завершения daemon-потоков при выходе из программы. Примеры: потоки сборщика мусора, потоки таймеров, потоки обработки событий в Swing. В Android daemon-потоки используются редко, так как приложение может быть убито системой в любой момент; вместо этого применяются Coroutines с правильной отменой или WorkManager для фоновых задач.

Что делают методы sleep(), yield(), join() и interrupt()?
Thread.sleep(millis) — приостанавливает текущий поток на указанное время, не освобождая монитор (если поток внутри synchronized). yield() — предлагает планировщику отдать процессорное время другому потоку того же приоритета (не гарантировано). join() — заставляет текущий поток ждать завершения другого потока (например, thread.join() ждёт окончания thread). interrupt() — устанавливает флаг прерывания потока; поток может проверить флаг через Thread.interrupted() или поймать InterruptedException (например, из sleep). В Android эти методы используются редко — предпочтительнее Coroutines с delay() и cancel().

Что такое ThreadLocal и где его используют в Android?
ThreadLocal — класс, который предоставляет переменную, уникальную для каждого потока: каждый поток видит своё собственное значение. При вызове set() значение сохраняется только для текущего потока, get() возвращает значение именно этого потока. Используется для хранения контекста, который должен быть разным в каждом потоке (например, локальный кэш, транзакция БД, пользовательская локаль). В Android классический пример — Looper и Handler, где каждый поток имеет свой Looper через ThreadLocal. Также может использоваться для хранения временных данных в пуле потоков (например, в ExecutorService).

Что такое Race Condition? Приведи реальный пример из Android.
Race Condition — ситуация, когда результат программы зависит от порядка выполнения потоков, а не от логики кода. Это происходит при одновременном доступе нескольких потоков к общей изменяемой переменной без синхронизации. Реальный пример в Android: счётчик нажатий кнопки var clickCount = 0. Два быстрых нажатия в разных потоках (например, через debounce и параллельный запрос) могут прочитать одно и то же значение, увеличить его и записать одинаковый результат — вместо +2 получится +1. Решается через synchronized, AtomicInteger или Mutex в Coroutines.

Что такое Deadlock? Как его предотвратить?
Deadlock — ситуация, когда два или более потока заблокировали друг друга, ожидая освобождения ресурсов, которые держит противоположный поток, и ни один не может продолжить выполнение. Классический пример — два потока, каждый из которых захватил один монитор и ждёт второй. Предотвращение: всегда захватывать ресурсы в одном и том же порядке, использовать таймауты на lock (tryLock()), применять ReentrantLock с fairness, избегать вложенных блокировок, использовать высокоуровневые механизмы (Coroutines, Channels). В Android deadlock часто возникает при неправильной синхронизации с UI (например, synchronized блок на Main Thread + ожидание фонового результата).

Что такое Livelock и Starvation?
Livelock — ситуация, когда потоки активно выполняют действия, но не могут продвинуться к цели из-за постоянной реакции друг на друга (например, два потока постоянно уступают друг другу дорогу, переходя в состояние ожидания). Starvation — когда поток или группа потоков постоянно откладываются и не получают процессорное время из-за более высокоприоритетных потоков или несправедливого планировщика. В Android livelock может возникнуть в сложных системах уведомлений, starvation — при высокой конкуренции за Main Thread или в пуле потоков с фиксированным размером.

Что такое Memory Visibility Problem и как с ним бороться?
Memory Visibility Problem — ситуация, когда один поток записал значение в переменную, но другой поток не видит это изменение из-за кэширования процессора или реордеринга инструкций. В Java/Kotlin без правильной синхронизации изменения могут быть невидимы для других потоков. Борьба: использование volatile (гарантирует видимость и запрещает реордеринг), synchronized, Atomic классы, Lock с volatile-полями, или высокоуровневые механизмы Coroutines с правильным диспетчером.

Что такое Happens-Before relationship?
Happens-Before — отношение в модели памяти Java, которое гарантирует, что операции одного потока видны другому потоку в определённом порядке. Основные правила: запись в volatile-поле happens-before чтение этого поля; выход из synchronized-блока happens-before вход в тот же блок; завершение потока happens-before join(); действия в конструкторе happens-before присвоение ссылки на объект. Это формальная гарантия видимости изменений без лишних блокировок. В Android это критично при работе с LiveData/StateFlow и многопоточными обновлениями UI.

Как работает ключевое слово synchronized?
Ключевое слово synchronized в Java (и Kotlin через @Synchronized или synchronized блок) обеспечивает взаимоисключающий доступ к критической секции кода, предотвращая одновременное выполнение этой секции несколькими потоками. При использовании на методе (synchronized fun method()) блокировка происходит на объекте this (для нестатических методов) или на классе (для статических). При использовании в блоке (synchronized(lock) { ... }) блокировка захватывается на указанном объекте-мониторе. Когда поток входит в synchronized-секцию, он захватывает монитор объекта; другие потоки, пытающиеся войти в эту же секцию, блокируются до освобождения монитора. После выхода из блока монитор автоматически освобождается, даже при исключении. Это гарантирует атомарность операций и видимость изменений памяти (happens-before).

Разница между synchronized методом и блоком?
Synchronized-метод автоматически оборачивает весь код метода в блокировку на объекте this (или классе для статического метода), что удобно, но может привести к избыточной длительности блокировки, если часть метода не требует синхронизации. Synchronized-блок позволяет точно указать объект-монитор (synchronized(lock) { ... }) и ограничить синхронизируемую область только критической секцией, что повышает производительность при высокой конкуренции. Блок также даёт гибкость: можно использовать разные объекты-мониторы для разных операций в одном классе. В Android рекомендуется использовать блок с явным лок-объектом (например, private val lock = Any()), чтобы избежать случайной блокировки на самом объекте класса, что может привести к неожиданным дедлокам.

Что такое volatile и когда его обязательно использовать?
volatile — модификатор поля, который гарантирует видимость изменений значения этого поля для всех потоков и запрещает компилятору и процессору выполнять реордеринг инструкций вокруг доступа к полю. Без volatile один поток может не увидеть изменения, сделанные другим потоком, из-за кэширования в локальном кэше процессора. volatile обеспечивает happens-before: запись в volatile-поле видна всем последующим чтениям этого поля. Обязательно использовать volatile для флагов завершения (volatile var running = true), простых статусов или ссылок, которые обновляются одним потоком и читаются другими, без дополнительных операций (например, volatile var latestData: Data?). Не заменяет полную синхронизацию при составных операциях (например, i++).

Что такое ReentrantLock, ReadWriteLock и когда их использовать вместо synchronized?
ReentrantLock — явная блокировка с возможностью повторного захвата тем же потоком (reentrant), поддержкой tryLock с таймаутом, fairness и interruptible lock. В отличие от synchronized, позволяет проверять состояние блокировки, прерывать ожидание и выбирать справедливый порядок. ReadWriteLock (обычно ReentrantReadWriteLock) разделяет блокировку на чтение (множественные читатели одновременно) и запись (эксклюзивно). Используют вместо synchronized, когда нужна тонкая блокировка: ReentrantLock — для сложной логики синхронизации (таймауты, fairness), ReadWriteLock — для сценариев «читать много, писать редко» (например, кэш данных в Android-приложении). В большинстве случаев synchronized проще и быстрее, но явные locks предпочтительнее при необходимости продвинутого контроля.

Что такое Atomic-классы (AtomicInteger, AtomicReference и т.д.)?
Atomic-классы из пакета java.util.concurrent.atomic предоставляют потокобезопасные операции над примитивными типами и ссылками без явной блокировки. Они используют Compare-And-Swap (CAS) на уровне процессора, что делает операции атомарными и неблокирующими. Примеры: AtomicInteger.incrementAndGet(), AtomicReference.compareAndSet(). Позволяют выполнять простые операции (get, set, increment, compareAndSet) без synchronized, что снижает overhead и повышает производительность при высокой конкуренции.

Чем Atomic лучше synchronized в некоторых случаях?
Atomic-классы обеспечивают атомарность без блокировки, используя lock-free алгоритмы на основе CAS, что исключает переключение контекста и ожидание в очереди блокировок. synchronized всегда использует монитор и может привести к приостановке потоков, особенно при высокой конкуренции. Atomic предпочтительнее для простых операций (счётчики, флаги, одиночные ссылки), где не требуется сложная логика внутри критической секции. В Android это критично для счётчиков нажатий, состояния загрузки или обновления UI-данных из нескольких корутин без лишних блокировок.

Что такое Coroutines и чем они принципиально отличаются от Threads?
Coroutines — это лёгковесный механизм асинхронного программирования в Kotlin, позволяющий писать асинхронный код в последовательном стиле с помощью suspend-функций. В отличие от потоков, корутины не привязаны к конкретному потоку ОС: они могут приостанавливаться и возобновляться на разных потоках, не блокируя их. Потоки — дорогостоящие (1 МБ стека, переключение контекста), их количество ограничено (тысячи). Корутины — дешёвые (несколько килобайт), их можно создавать десятками тысяч. В Android корутины позволяют писать асинхронный код без callback hell, с автоматической отменой при уничтожении экрана и поддержкой Structured Concurrency.

Что такое suspend функция и как она работает под капотом?
suspend — модификатор функции, указывающий, что она может приостанавливаться (suspend) без блокировки потока. Под капотом компилятор преобразует suspend-функцию в Continuation-passing style: добавляется параметр Continuation, который хранит состояние приостановки. При вызове suspend функция возвращает результат через Continuation.resume(). Это позволяет библиотекам (например, Retrofit, Room) приостанавливать выполнение на время ожидания I/O, освобождая поток. В отличие от обычных функций, suspend-функции можно вызывать только из других suspend-функций или корутин-билдеров (launch, async).

Что такое Coroutine Context и Coroutine Scope?
Coroutine Context — это набор элементов (Job, Dispatcher, CoroutineName, ExceptionHandler и т.д.), определяющих поведение корутины: на каком потоке выполнять, как обрабатывать исключения, есть ли родительская Job. Coroutine Scope — это область видимости корутин, которая управляет жизненным циклом всех дочерних корутин через Job. Scope гарантирует, что все дочерние корутины будут отменены при отмене scope (Structured Concurrency). В Android viewModelScope и lifecycleScope — это готовые scope, автоматически отменяющиеся при уничтожении ViewModel или Fragment.

Какие встроенные Coroutine Scopes существуют в Android?
В Android существуют готовые scope: lifecycleScope (привязан к жизненному циклу Activity/Fragment), viewModelScope (привязан к ViewModel, живёт дольше конфигурационных изменений), repeatOnLifecycle (для Flow внутри lifecycle-aware компонентов). Также часто создают свои scope на основе CoroutineScope(SupervisorJob() + Dispatchers.Default) для фоновых задач. GlobalScope существует, но не рекомендуется в продакшене, так как не привязан к жизненному циклу и может привести к утечкам.

Разница между viewModelScope, lifecycleScope и GlobalScope?
viewModelScope привязан к жизненному циклу ViewModel: отменяется при onCleared(), идеален для операций, связанных с данными экрана. lifecycleScope привязан к жизненному циклу Activity/Fragment, отменяется при ON_DESTROY, подходит для UI-задач. GlobalScope — глобальный scope без автоматической отмены, может привести к утечкам корутин после уничтожения экрана. Правило: используйте viewModelScope для логики ViewModel, lifecycleScope для задач внутри Activity/Fragment, избегайте GlobalScope.

Что такое Structured Concurrency и почему это важно?
Structured Concurrency — принцип, при котором все дочерние корутины должны завершиться до завершения родительской корутины или scope. Это достигается через Coroutine Scope и Job: при отмене scope отменяются все дочерние корутины, при исключении в дочерней корутине оно передаётся вверх. Важно, потому что предотвращает утечки корутин, упрощает управление жизненным циклом и гарантирует, что ресурсы освобождаются. В Android это критично: без Structured Concurrency корутины могут продолжать работать после уничтожения экрана, вызывая утечки памяти и краши.

Разница между coroutineScope и supervisorScope?
coroutineScope — обычный scope, который отменяет все дочерние корутины при любом исключении и передаёт исключение вверх (fail-fast). supervisorScope — scope, который не отменяет остальные дочерние корутины при исключении в одной из них и не передаёт исключение вверх (fail-safe). Используется, когда сбой одной задачи не должен останавливать остальные (например, параллельная загрузка нескольких изображений). В Android supervisorScope полезен в ViewModel для независимых фоновых задач.

Что такое Job и Deferred?
Job — объект, представляющий жизненный цикл корутины: может быть активным, завершённым, отменённым. Позволяет отменять корутину (job.cancel()) и отслеживать состояние. Deferred — это Job с результатом: возвращается из async, позволяет получить результат через await(). async возвращает Deferred<T>, launch — просто Job. В Android Job используется для отмены задач при уничтожении экрана, Deferred — для получения результата асинхронных вычислений.

Как правильно отменять корутины?
Отмена корутин выполняется через Job.cancel() или scope.cancel(). Корутина должна быть кооперативной: проверять isActive или использовать ensureActive(), yield(), delay(). При использовании withContext, withTimeout отмена передаётся автоматически. Лучшая практика — привязывать корутины к viewModelScope или lifecycleScope, которые автоматически отменяются при уничтожении компонента. Вручную отменять нужно редко, но если корутина не кооперативна — отмена не сработает.

Что такое withContext, withTimeout, withTimeoutOrNull?
withContext(Dispatcher) переключает контекст выполнения на указанный диспетчер (например, с Main на IO) и ждёт завершения блока. withTimeout(timeMillis) ограничивает время выполнения блока и бросает TimeoutCancellationException при превышении. withTimeoutOrNull возвращает null вместо исключения при таймауте. Все три функции отменяемы и поддерживают Structured Concurrency. В Android withContext(Dispatchers.IO) используется для тяжёлой работы, withTimeout — для ограничения долгих операций.

Какие Dispatchers существуют? Когда какой использовать?
Основные диспетчеры: Dispatchers.Main — для работы с UI (обновление View, LiveData.postValue). Dispatchers.IO — для дисковых операций, сетевых запросов, Room-запросов (ограниченный пул потоков). Dispatchers.Default — для CPU-интенсивных задач (парсинг, вычисления), использует пул потоков по числу ядер. Dispatchers.Unconfined — выполняется в текущем потоке без переключения (редко, опасно). Dispatchers.Main.immediate — выполняется немедленно, если уже на Main Thread. В Android Main — для UI, IO — для I/O, Default — для вычислений.

Почему нельзя использовать Dispatchers.Main для тяжёлой работы?
Dispatchers.Main привязан к Main Thread (UI Thread), и выполнение длительных операций в нём блокирует отрисовку интерфейса, обработку событий и анимации, что приводит к зависанию UI и ANR. Android строго запрещает тяжёлую работу (сеть, БД, парсинг больших JSON) на Main Thread, выбрасывая исключения (NetworkOnMainThreadException). Все задачи, требующие ожидания или вычислений, должны переключаться на IO или Default через withContext.

Что такое Dispatchers.IO vs Dispatchers.Default?
Dispatchers.IO предназначен для операций ввода-вывода (сеть, диск, БД), использует ограниченный пул потоков (по умолчанию до 64 или число ядер × 64), чтобы не перегружать систему. Dispatchers.Default предназначен для CPU-интенсивных задач (парсинг, шифрование, вычисления), использует пул потоков по числу ядер процессора. В Android IO — для Retrofit, Room, файлов; Default — для сложных вычислений в фоне.

Можно ли запускать корутины на Dispatchers.Main.immediate?
Да, Dispatchers.Main.immediate позволяет запускать корутину немедленно, если текущий поток уже Main Thread, без постановки в очередь. Это полезно для микрозадач, которые должны выполниться синхронно на UI-потоке (например, немедленное обновление View после события). Обычный Dispatchers.Main всегда ставит задачу в очередь, даже если уже на Main. Однако Main.immediate следует использовать осторожно: если задача тяжёлая, она всё равно заблокирует UI.

Что такое Kotlin Flow?
Kotlin Flow — это асинхронный поток данных в Kotlin Coroutines, представляющий собой последовательность значений, которые могут быть вычислены асинхронно и доставлены подписчикам по мере их появления. Flow является холодным (cold) по умолчанию: он начинает производить значения только при наличии активного коллектора (collect) и завершается вместе с ним. Flow поддерживает операторы для трансформации (map, filter), комбинирования (combine, zip) и обработки ошибок, что делает его мощным инструментом для реактивного программирования. В Android Flow широко используется для работы с Room (как Flow-результат запроса), Retrofit (через flow-адаптер), StateFlow для UI-состояния и SharedFlow для событий.

Разница между Cold Flow и Hot Flow?
Cold Flow запускает производство данных заново для каждого нового коллектора: каждый collect вызывает выполнение блока flow { … } с нуля. Это означает, что если два коллектора подписываются на один и тот же cold Flow, каждый получит независимый набор значений. Hot Flow (StateFlow, SharedFlow) производит данные независимо от количества коллекторов: эмиссия начинается один раз (обычно при создании или при первом подписчике), и все коллекторы получают одни и те же значения, включая уже существующие (в случае replay). Cold Flow подходит для одноразовых запросов (например, сетевой вызов), hot Flow — для общего состояния или событий (UI-данные, уведомления).

Что такое StateFlow и SharedFlow?
StateFlow — это горячий поток, который всегда хранит последнее значение и сразу выдаёт его новому коллектору (аналогично LiveData). Он имеет начальное значение, поддерживает distinctUntilChanged по умолчанию и предназначен для представления состояния (например, UI state: loading/data/error). SharedFlow — более общий горячий поток без обязательного начального значения, может иметь replay (количество последних значений для новых подписчиков) и используется для одноразовых событий (например, показ Snackbar, навигация, toast). StateFlow всегда имеет ровно одно актуальное значение, SharedFlow может не иметь значения или иметь несколько.

Когда использовать StateFlow, а когда SharedFlow?
StateFlow используется для представления состояния экрана или данных, которые всегда должны иметь актуальное значение (например, список пользователей, статус загрузки, выбранный элемент). Новый наблюдатель сразу получает текущее состояние без дополнительных запросов. SharedFlow применяется для событий и уведомлений, которые происходят один раз и не требуют хранения значения (показ сообщения, навигация, аналитика). Если нужно, чтобы новые подписчики получили предыдущие события — задаём replay > 0, иначе SharedFlow подходит для fire-and-forget событий.

Что такое shareIn и stateIn?
shareIn и stateIn — операторы, превращающие cold Flow в hot (shared) поток с управляемым жизненным циклом. shareIn(scope, replay) преобразует Flow в SharedFlow с заданным replay и автоматической остановкой при отмене scope. stateIn(scope, initialValue) превращает Flow в StateFlow с начальным значением и автоматической подпиской в scope. Оба оператора обеспечивают, что эмиссия начинается только при наличии коллекторов и останавливается при отмене scope, предотвращая утечки. В Android их используют для преобразования Room Flow или сетевых запросов в StateFlow для UI.

Как правильно использовать repeatOnLifecycle с Flow?
repeatOnLifecycle — расширение из lifecycle-runtime-ktx, позволяющее собирать Flow только в определённых состояниях жизненного цикла (например, STARTED или RESUMED). Синтаксис: lifecycleScope.launch { repeatOnLifecycle(Lifecycle.State.STARTED) { flow.collect { … } } }. Это автоматически отменяет сбор при уходе в STOPPED и возобновляет при возвращении, предотвращая лишнюю работу и утечки. В Android это стандартный способ подписки на Flow в Fragment/Activity, чтобы не собирать данные, когда экран невидим.

Что такое combine, zip, flatMapConcat, flatMapLatest, debounce, distinctUntilChanged?
combine объединяет несколько Flow, выдавая значение при изменении любого из них (например, combine(userFlow, settingsFlow)). zip ждёт значение от каждого Flow и выдаёт результат только при наличии всех. flatMapConcat последовательно обрабатывает внутренние Flow (ждёт завершения предыдущего). flatMapLatest отменяет предыдущий внутренний Flow при новом значении внешнего (актуально для поиска). debounce подавляет частые эмиссии, выдавая последнее значение через заданное время. distinctUntilChanged фильтрует последовательные одинаковые значения.

Как тестировать Flow и StateFlow?
Flow и StateFlow тестируются с помощью kotlinx-coroutines-test: runTest создаёт тестовый scope с виртуальным временем. Для Flow используют collect в тесте или библиотеку Turbine (flow.test { expectItem(), expectNoEvents() }). Для StateFlow проверяют начальное значение и последующие через value или collect. advanceUntilIdle() продвигает время до завершения всех задач, advanceTimeBy() симулирует задержку. Тесты пишутся в runTest с TestScope и StandardTestDispatcher.

Как тестировать код с корутинами?
Тестирование корутин проводится с помощью kotlinx-coroutines-test. Основной инструмент — runTest, который создаёт контролируемый TestScope и TestDispatcher. Внутри runTest можно использовать advanceUntilIdle() для завершения всех задач, advanceTimeBy() для симуляции времени, runCurrent() для выполнения отложенных задач. Для suspend-функций тестируют через runTest { mySuspendFun() }. Для отмены проверяют ensureActive() или исключения при отмене scope.

Что такое runTest, TestScope, StandardTestDispatcher?
runTest — тестовая функция-обёртка, создающая TestScope и StandardTestDispatcher для управления временем и корутинами. TestScope — тестовый scope, автоматически отменяющийся после теста. StandardTestDispatcher — виртуальный диспетчер, позволяющий контролировать выполнение задач через advanceUntilIdle(), advanceTimeBy(), runCurrent(). Это позволяет детерминированно тестировать задержки, таймауты и последовательность корутин.

Как использовать Turbine для тестирования Flow?
Turbine — библиотека для удобного тестирования Flow. Синтаксис: flow.test { awaitItem(), expectItem(expected), expectNoEvents(), awaitComplete() }. Позволяет проверять последовательность эмиссий, ошибки, завершение и отсутствие событий. Работает с runTest и TestScope, поддерживает expectMostRecentItem(), expectNoMoreEvents(). Идеально для тестирования StateFlow, SharedFlow и сложных цепочек операторов.

Как тестировать StateFlow и SharedFlow?
StateFlow тестируют через прямой доступ к value или collect в runTest: проверяют начальное значение и последующие эмиссии. SharedFlow тестируют через collect с Turbine или прямой collect с expectItem(). Для обоих используют advanceUntilIdle() после эмиссий, чтобы дождаться обработки. В ViewModel-тестах проверяют, что StateFlow обновляется при вызове методов и сохраняет последнее значение.

Что такое advanceUntilIdle() и advanceTimeBy()?
advanceUntilIdle() продвигает виртуальное время до выполнения всех отложенных задач в TestDispatcher, имитируя завершение всех текущих корутин. advanceTimeBy(millis) продвигает время на указанное количество миллисекунд, позволяя протестировать delay(), withTimeout и debounce. В тестах это ключевые методы для управления временем и проверки поведения корутин с задержками.

Coroutines vs RxJava — что лучше в 2025 году и почему?
В 2025 году Kotlin Coroutines являются предпочтительным выбором для большинства Android-приложений по сравнению с RxJava. Coroutines предлагают более естественный, последовательный стиль написания асинхронного кода благодаря suspend-функциям, что значительно снижает сложность по сравнению с цепочками операторов RxJava. Они имеют встроенную поддержку Structured Concurrency, автоматическую отмену при уничтожении экрана (viewModelScope, lifecycleScope), меньший overhead и лучшую интеграцию с Jetpack-библиотеками (Room, Retrofit, Paging, DataStore). RxJava остаётся актуальной только в legacy-проектах или в случаях, когда требуется сложная обработка потоков с backpressure (например, в высоконагруженных системах с потоками данных). Coroutines проще тестировать (runTest + Turbine), имеют меньший размер APK и лучше поддерживаются сообществом Google.

Coroutines vs Threads — когда всё-таки лучше использовать обычные Thread?
Обычные Thread следует использовать крайне редко в современных Android-приложениях, только в случаях, когда требуется низкоуровневый контроль над потоками ОС или интеграция с legacy-кодом, использующим Thread напрямую (например, старые библиотеки с собственным пулами потоков). Coroutines значительно легче, дешевле по памяти, поддерживают Structured Concurrency и автоматическую отмену, а также позволяют писать асинхронный код в синхронном стиле. Thread не имеют встроенной отмены, сложнее отлаживать и могут привести к утечкам, если забыть остановить. В 2025 году единственные обоснованные сценарии для Thread — это высокопроизводительные нативные вызовы (JNI) или специальные задачи, где Coroutines по каким-то причинам не подходят.

AsyncTask vs Coroutines vs WorkManager — когда что использовать?
AsyncTask полностью устарел и не рекомендуется с 2018 года (deprecated в Android 11+): он не поддерживает отмену, легко приводит к утечкам и не учитывает жизненный цикл. Coroutines — основной инструмент для асинхронных задач внутри приложения (сетевые запросы, работа с БД, вычисления), особенно с viewModelScope и lifecycleScope — они автоматически отменяются при уничтожении экрана и хорошо интегрируются с Jetpack. WorkManager предназначен для отложенных, гарантированных фоновых задач, которые должны выполниться даже после перезапуска приложения (например, синхронизация данных, загрузка файлов, отправка аналитики), поддерживает ограничения (сеть, заряд батареи) и перезапуск при сбоях. Простое правило: Coroutines — для задач внутри экрана, WorkManager — для задач вне экрана.

Как реализовать параллельную загрузку нескольких изображений?
Параллельная загрузка изображений в Kotlin Coroutines реализуется через async внутри coroutineScope или supervisorScope с последующим awaitAll. Пример: supervisorScope { val deferreds = urls.map { url -> async(Dispatchers.IO) { loadImage(url) } }; val images = deferreds.awaitAll() }. supervisorScope обеспечивает, что сбой загрузки одного изображения не отменяет остальные. Для обновления UI можно использовать withContext(Dispatchers.Main) или StateFlow. Это эффективнее последовательной загрузки, использует пул потоков IO и легко отменяется при уничтожении экрана.

Как правильно использовать Mutex и Semaphore в Coroutines?
Mutex — это корутинный аналог ReentrantLock, используется для защиты критической секции в корутинах: mutex.withLock { критический код }. Поддерживает отмену и не блокирует поток. Semaphore ограничивает количество одновременно выполняемых задач (например, 3 параллельных запроса): semaphore.withPermit { ... }. В Android Mutex часто применяют для защиты общей переменной или очереди в ViewModel, Semaphore — для ограничения параллельных сетевых запросов. Важно: использовать их только внутри suspend-функций, избегать вложенных блокировок и всегда учитывать Structured Concurrency.

Что такое Channel и когда его использовать вместо Flow?
Channel — это потокобезопасная очередь сообщений между корутинами (аналог BlockingQueue, но для suspend). Поддерживает send/receive, capacity, onBufferOverflow (SUSPEND, DROP_OLDEST, DROP_LATEST). Channel используется, когда нужно передавать события между корутинами в стиле producer-consumer без хранения состояния (например, очередь задач, события от сервиса). Flow подходит для потоков данных с операторами и реактивной обработкой. Channel предпочтительнее, когда требуется точный контроль над очередью и backpressure (например, send не должен блокировать, если буфер полон).

Разница между produce, actor и обычными корутинами?
produce создаёт потокобезопасный producer-канал, который отправляет элементы в Channel и автоматически закрывает его при завершении. actor — это корутина, которая обрабатывает сообщения из Channel последовательно, гарантируя thread-safety без дополнительных блокировок (аналог actor-модели). Обычные корутины (launch, async) не имеют встроенного канала и не гарантируют последовательной обработки сообщений. Actor и produce полезны для высоконагруженных сценариев с большим количеством событий, где нужна строгая последовательность.

Как сделать thread-safe singleton в Kotlin?
Самый простой и рекомендуемый способ — использовать object (Kotlin singleton): object MySingleton { ... }. Он инициализируется лениво, потокобезопасен и гарантирует единственный экземпляр. Альтернатива — val instance by lazy(LazyThreadSafetyMode.SYNCHRONIZED) { MyClass() } или double-checked locking с volatile в Java-стиле. В Android object — стандарт для App-level синглтонов (например, аналитика, логгер). Избегайте статических полей с mutable состоянием.

Как работать с SharedPreferences в многопоточной среде?
SharedPreferences в Android потокобезопасны для чтения, но запись через apply() асинхронна, а commit() синхронна и блокирует поток. При параллельной записи из нескольких потоков возможны race condition и потеря данных. Рекомендуется использовать edit().put…().apply() в одном потоке (например, через Dispatchers.IO в корутине) или Mutex для защиты секции редактирования. Для частых операций лучше перейти на DataStore (Preferences DataStore), который изначально потокобезопасен и поддерживает корутины.

Как безопасно обновлять UI из нескольких корутин?
Обновление UI должно происходить только на Dispatchers.Main. Используйте withContext(Dispatchers.Main) или launch(Dispatchers.Main) внутри ViewModel. Лучшая практика — собирать данные в StateFlow или LiveData в ViewModel, а UI (Activity/Fragment) наблюдает за ним через collectAsState() или observe(). Это гарантирует, что все обновления происходят в правильном потоке и учитывают жизненный цикл. Избегайте прямых вызовов runOnUiThread — Coroutines + StateFlow делают это автоматически и безопасно.

Как отлаживать зависшие корутины и deadlocks?
Для отладки зависших корутин используют Android Studio Profiler (CPU → Threads) для просмотра стека потоков, Thread Dump (через ADB или Profiler) для анализа блокировок и kotlinx-coroutines-debug для вывода состояния всех корутин (Coroutine Debugger). Deadlock обнаруживают через Thread Dump (ищут потоки в состоянии WAITING/BLOCKED на одном мониторе) или Profiler. Для корутин — логирование Job состояния, использование CoroutineExceptionHandler и проверка isActive/ensureActive(). В продакшене добавляют Crashlytics/ Sentry с Thread Dump.