

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 (создается анонимный класс), что дорого по памяти и времени.

С inline — лямбда встраивается как обычный код, и объекта не создается.

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.

Требуется 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)`.

Требуется `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 инстанцирован 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 ОСП 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 ($\Omega(f(n))$) — нижнюю границу, лучший случай или асимптотическую нижнюю оценку. Theta ($\Theta(f(n))$) — 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: $\text{base_address} + \text{index} * \text{element_size}$, 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 $O(n^2)$.

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, $O(n^2)$.

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 ($O(n^2)$ на большом n) блокируют главный поток, вызывая ANR (Application Not Responding) при задержке >5 секунд.

Почему $O(n^2)$ алгоритмы опасны в onDraw или RecyclerView?

В onDraw или onBindViewHolder $O(n^2)$ 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.

Coroutines

Suspend fun — это просто функция, которую можно вызывать только из корутины или другой suspend-функции. Она может «зависать» без блокировки потока (например, ждать сеть или базу). Это не стрим, это разовый вызов.

Flow — холодный стрим. Каждый раз, когда кто-то делает collect, весь код внутри flow { ... } выполняется заново. Если никто не подписан — ничего не происходит. Идеально для запросов к базе, сети, чтения файла — каждый подписчик получает свои данные с нуля.

Channel — горячий стрим старого образца (почти устарел). Код запускается один раз, и данные идут в один канал. Если потребителей несколько — они делят одну очередь, кто первый успел — тот получил. Если никто не слушает — данные теряются (кроме буферизированных вариантов).

SharedFlow — современный горячий стрим. Код внутри запускается один раз, но подписаться могут сколько угодно потребителей одновременно. Можно настроить replay (сколько последних значений запоминать). Отлично подходит для событий: клик по кнопке, навигация, показ тоста/snackbar — то, что не нужно повторять при новой подписке.

StateFlow — это специальный случай SharedFlow с replay = 1 и политикой «всегда есть актуальное значение». Он всегда хранит последнее состояние и отдаёт его сразу при новой подписке. Это золотой стандарт для UI-состояния в Android: loading/error/data, поисковый запрос, выбранный таб, чекбоксы и т.д.