

# 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 генерируются автоматически.

Эти методы генерируются только по свойствам из основного (primary) конструктора.

**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 nonnull. **!!**

— 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