

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 (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.

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.

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