Ссылка на видео этого руководства на сайте: borisproit.expert

A. Kotlin базовые концепции

Что такое null safety в Kotlin? What is null safety in Kotlin?

Null safety — это механизм Kotlin, который позволяет предотвратить ошибки типа NullPointerException за счёт явного разделения типов на nullable и non-nullable.
Переменная не может быть null, если это не указано явно через `?`.

Null safety is a Kotlin feature that prevents NullPointerException by distinguishing between nullable and non-nullable types.
A variable cannot be null unless it is explicitly declared with `?`.

```
var name: String = "Alex"      // non-nullable type
// name = null ❌ compile error

var nickname: String? = null   // nullable type

println(nickname?.length)      // safe call
```

Чем отличается val от var? What is the difference between val and var?

`val` — это неизменяемая ссылка. Значение можно присвоить только один раз.
`var` — это изменяемая ссылка. Значение можно менять.

`val` is an immutable reference. It can be assigned only once.
`var` is a mutable reference. Its value can be changed.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
val name = "Alex"      // immutable reference
// name = "John" ❌ compile error

var age = 25           // mutable reference
age = 26               // allowed
```

Чем отличается val от const val? What is the difference between val and const val?

`val` — это неизменяемая переменная, значение которой может быть известно только во время выполнения.
`const val` — это константа, значение которой известно во время компиляции.

`val` is an immutable variable whose value is known at runtime.
`const val` is a compile-time constant whose value is known during compilation.

```
val runtimeValue = System.currentTimeMillis()  // evaluated at runtime

const val MAX_USERS = 100                      // compile-time constant
```

Что такое type inference в Kotlin? What is type inference in Kotlin?

Type inference — это способность Kotlin автоматически определять тип переменной.

Type inference is Kotlin's ability to automatically determine the type of a variable.

```
val name = "Alex"   // inferred as String

println(name)   // prints Alex
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Что делает оператор ?. ? What does the ?. operator do?

Оператор ?. выполняет безопасный вызов.
Он вызывает метод или обращается к свойству только если объект не равен null.

The ?. operator performs a safe call.
It calls a method or accesses a property only if the object is not null.

```
var name: String? = null

println(name?.length)   // safe call, returns null instead of crashing
```

Что такое smart cast? What is smart cast?

Smart cast — это автоматическое приведение типа после проверки.

Smart cast is automatic type casting after a check.

```
fun printLength(value: Any) {
    if (value is String) {
        println(value.length)   // prints length
    }
}

printLength("Hello")   // prints 5
```

Чем when отличается от switch?
What is the difference between when and switch?

Ссылка на видео этого руководства на сайте: borisproit.expert

`switch` — это конструкция из Java, которая работает только с ограниченными типами и требует break.
`when` — более гибкая конструкция, которая работает с любыми типами, поддерживает условия и не требует break.

`switch` in Java works with limited types and requires break.
`when` is more flexible, works with any type, supports conditions, and does not require break.

```
val number = 2

when (number) {
    1 -> println("One")
    2 -> println("Two")   // prints Two
    else -> println("Other")
}
```

Что делает оператор `!!` и почему его не любят? What does the `!!` operator do and why is it disliked?

Оператор `!!` принудительно преобразует nullable тип в non-nullable.
Если значение окажется null — произойдёт NullPointerException.

The `!!` operator forcefully converts a nullable type into a non-nullable type.
If the value is null — it throws a NullPointerException.

```
var name: String? = null

println(name!!.length)   // force unwrap, throws NullPointerException
```

Что делает Elvis оператор `?:` ? What does the Elvis operator `?:` do?

Ссылка на видео этого руководства на сайте: borisproit.expert

Elvis оператор возвращает значение слева, если оно не равно null, иначе — значение справа.

The Elvis operator returns the value on the left if it is not null, otherwise it returns the value on the right.

```
var name: String? = null

val length = name?.length ?: 0   // fallback value if null
println(length)
```

Чем отличается null от empty?
What is the difference between null and empty?

`null` означает отсутствие значения.
`empty` означает наличие значения, но без содержимого.
Используйте `null`, когда значение отсутствует.
Используйте `empty`, когда объект существует, но пуст.

`null` means no value exists.
`empty` means a value exists but contains nothing.
Use `null` when the value is absent.
Use `empty` when the object exists but has no content.

```
val name: String? = null
val list = emptyList<Int>()

println(name)   // prints null
println(list)   // prints []
```

Чем отличается `==` от `===` ? What is the difference between `==` and `===`?

Ссылка на видео этого руководства на сайте: borisproit.expert

== проверяет структурное равенство (сравнивает значения).
=== проверяет ссылочное равенство (сравнивает, указывают ли переменные на один объект).

== checks structural equality (compares values).
=== checks referential equality (compares if two references point to the same object).

```
val a = String("Hello".toCharArray())
val b = String("Hello".toCharArray())

println(a == b)    // true - same value
println(a === b)   // false - different objects
```

Что такое data class и зачем он нужен? What is a data class and why is it needed?

Data class — это класс, предназначенный для хранения данных.
Он автоматически генерирует equals(), hashCode(), toString(), copy() и componentN().

A data class is a class designed to hold data.
It automatically generates equals(), hashCode(), toString(), copy(), and componentN().

```
data class User(val name: String, val age: Int)

val user1 = User("Alex", 25)
val user2 = user1.copy(age = 26)   // copy with change

println(user1)   // auto toString
```

Что такое sealed class и где его используют? What is a sealed class and where is it used?

Ссылка на видео этого руководства на сайте: borisproit.expert

Sealed class — это ограниченная иерархия классов, где все возможные наследники известны заранее.
Чаще всего используется для моделирования состояний (UI state, Result).

A sealed class is a restricted class hierarchy where all possible subclasses are known in advance.
It is commonly used to model states (UI state, Result).

```
sealed class UiState {
    object Loading : UiState()
    data class Success(val data: String) : UiState()
    data class Error(val message: String) : UiState()
}

fun render(state: UiState) {
    when(state) {
        is UiState.Loading -> println("Loading")
        is UiState.Success -> println(state.data)
        is UiState.Error -> println(state.message)
    }
}
```

Ссылка на видео этого руководства на сайте: borisproit.expert

В чём разница между class, data class, sealed class и enum?
What is the difference between class, data class, sealed class and enum?

`class` — обычный класс для логики и состояния.
`data class` — класс для хранения данных с авто-методами.
`sealed class` — ограниченная иерархия для описания состояний.
`enum` — фиксированный набор значений.

`class` — general-purpose class for logic and state.
`data class` — class for holding data with auto-generated methods.
`sealed class` — restricted hierarchy for modeling states.
`enum` — fixed set of values.

```
class User(val name: String)

data class Person(val name: String)

sealed class Result {
    object Success : Result()
    object Error : Result()
}

enum class Direction {
    LEFT, RIGHT
}

println(Direction.LEFT)   // prints LEFT
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Что такое value class?
What is a value class?

Value class — это класс-обёртка над одним значением без создания дополнительного объекта.
Лучше использовать, когда нужно добавить смысл к примитивному типу без лишних затрат.
Не стоит использовать, если нужно хранить несколько свойств или сложную логику.

A value class is a wrapper around a single value without creating an extra object.
Use it when you want to give meaning to a primitive type without extra overhead.
Avoid using it when you need multiple fields or complex logic.

```
@JvmInline
value class UserId(val id: Int)

fun printUser(userId: UserId) {
    println(userId.id)   // prints 10
}

printUser(UserId(10))
```

В чём разница между object declaration и object expression?

What is the difference between object declaration and object expression?

Object declaration — это именованный singleton.
Object expression — это анонимный объект, создаётся каждый раз.

Object declaration

Ссылка на видео этого руководства на сайте: borisproit.expert

```
object Logger {
    fun log() {}
}
```

Характеристики:

- имеет имя
- создаётся один раз
- лениво (при первом доступе)
- singleton

Используется для:

- глобальных сервисов
- утилит
- shared состояния

```
val listener = object : Runnable {
    override fun run() {}
}
```

Характеристики:

- без имени
  создаётся сразу
- новый экземпляр каждый раз
- часто используется как inline реализация

Используется для:

- callbacks
- listeners
- временной логики

Ссылка на видео этого руководства на сайте: borisproit.expert

Чем interface отличается от abstract class?
What is the difference between an interface and an abstract class?

`interface` описывает поведение и может иметь несколько реализаций.
`abstract class` может хранить состояние и предоставляет базовую реализацию.

An `interface` defines behavior and supports multiple implementations.
An `abstract class` can hold state and provide base implementation.

```
interface Logger {
    fun log()
}

abstract class BaseLogger {
    val prefix = "Log:"
    abstract fun log()
}

class FileLogger : BaseLogger() {
    override fun log() {
        println(prefix)   // prints Log:
    }
}
```

Могут ли интерфейсы иметь реализацию в Kotlin?
Can interfaces have implementation in Kotlin?

Да, интерфейсы могут иметь реализацию методов по умолчанию.

Ссылка на видео этого руководства на сайте: borisproit.expert

---

Yes, interfaces can have default method implementations.

```
interface Logger {
    fun log() {
        println("Default log")   // prints Default log
    }
}

class ConsoleLogger : Logger

ConsoleLogger().log()
```

Что такое primary и secondary constructor?
What are primary and secondary constructors?

Primary constructor — основной конструктор, объявляется в заголовке класса.
Secondary constructor — дополнительный конструктор, объявляется внутри класса.

Primary constructor is the main constructor declared in the class header.
Secondary constructor is an additional constructor declared inside the class.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
class User(val name: String) {

    constructor(name: String, age: Int) : this(name) {
        println("Age: $age")   // prints Age: 25
    }
}

val user = User("Alex", 25)
```

Когда использовать init, а когда secondary constructor?
When to use init vs secondary constructor?

`init` используют для общей логики инициализации.
Secondary constructor используют, когда нужен альтернативный способ создания объекта.

Use `init` for common initialization logic.
Use a secondary constructor when an alternative way of creating the object is needed.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
class User(val name: String) {

    init {
        println("User created")   // prints User created
    }

    constructor(name: String, age: Int) : this(name) {
        println("Age: $age")   // prints Age: 25
    }
}

val user = User("Alex", 25)
```

Что такое `object`? What is `object`?

`object` — это способ объявить singleton в Kotlin.
Он создаёт единственный экземпляр класса.

`object` is a way to declare a singleton in Kotlin.
It creates a single instance of a class.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
object Logger {
    fun log(message: String) {
        println(message)
    }
}

Logger.log("Hello")   // using the single instance
```

Что такое companion object? What is a companion object?

Companion object — это объект внутри класса, который используется для хранения статических членов.

A companion object is an object inside a class used to hold static members.

```
class User(val name: String) {
    companion object {
        fun createGuest() = User("Guest")
    }
}

val guest = User.createGuest()   // static-like call
```

Какая разница между `object` и `companion object`? What is the difference between `object` and `companion object`?

`object` — это отдельный singleton-класс.
`companion object` — это singleton внутри класса, который используется для статического доступа к членам класса.

Ссылка на видео этого руководства на сайте: borisproit.expert

---

`object` is a standalone singleton class.
`companion object` is a singleton inside a class used for static-like access to its members.

```
object Logger {
    fun log(message: String) = println(message)
}

class User {
    companion object {
        fun createGuest() = User()
    }
}

Logger.log("Hello")       // standalone singleton
User.createGuest()        // static-like access
```

Что такое lateinit и когда его можно использовать? What is `lateinit` and when can it be used?

`lateinit` — это модификатор для non-null переменных, которые будут инициализированы позже.
Используется, когда значение нельзя задать сразу (например, DI или View).

`lateinit` is a modifier for non-null variables that will be initialized later.
It is used when the value cannot be assigned immediately (e.g., DI or View).

Ссылка на видео этого руководства на сайте: borisproit.expert

```
lateinit var repository: UserRepository   // initialized later

fun init() {
    repository = UserRepository()         // initialization happens here
}
```

B. Функции

Что такое функции высшего порядка (Higher Order Functions)? What are Higher Order Functions?

Функции высшего порядка — это функции, которые принимают другие функции как параметр или возвращают функцию.

Higher order functions are functions that take other functions as parameters or return a function.

```
fun calculate(a: Int, b: Int, operation: (Int, Int) -> Int): Int {
    return operation(a, b)   // function as parameter
}

val result = calculate(2, 3) { x, y -> x + y }   // passing lambda
println(result)
```

```
operation: (Int, Int) -> Int
```

“Мне передадут функцию
которая принимает 2 числа
и возвращает число”

```
val result = calculate(2, 3) { x, y -> x + y }
```

```
{ x, y -> x + y }
```

"Возьми два числа сложи их"

Ссылка на видео этого руководства на сайте: borisproit.expert

Что такое lambda? What is a lambda?

Lambda — это анонимная функция, которую можно передать как значение.

A lambda is an anonymous function that can be passed as a value.

```
val sum = { a: Int, b: Int -> a + b }   // lambda function

println(sum(2, 3))   // calls lambda
```

В чём преимущество lambda? What is the advantage of lambda?

Lambda позволяет передавать поведение без создания отдельной функции или класса.
Это делает код короче и удобнее.

Lambda allows passing behavior without creating a separate function or class.
It makes the code shorter and more expressive.

```
fun sum(a: Int, b: Int) = a + b
fun multiply(a: Int, b: Int) = a * b
fun max(a: Int, b: Int) = if (a > b) a else b
```

Ссылка на видео этого руководства на сайте: borisproit.expert

```
fun calculate(a: Int, b: Int, operation: (Int, Int) -> Int): Int {
    return operation(a, b)
}

calculate(2, 3, ::sum)
```

- много мелких функций
- мусор в коде
- неудобно

```
calculate(2, 3) { x, y -> x + y }
calculate(2, 3) { x, y -> x * y }
calculate(2, 3) { x, y -> maxOf(x, y) }
```

Lambda = быстрый способ передать поведение

Пример:

```
users.filter { it.age > 18 }
```

Без lambda пришлось бы писать отдельную функцию.

Итог:
✔️ гибкость
✔️ меньше лишних функций
✔️ возможность менять поведение "на лету"

Но есть проблемы:

- объект в памяти
- который создаётся при вызове

Что такое inline функция? What is an inline function?

Inline функция — это функция, код которой подставляется в место вызова во время компиляции.

Ссылка на видео этого руководства на сайте: borisproit.expert

An inline function is a function whose code is inserted at the call site during compilation.

```
listOf(1,2,3).forEach {
    println(it)
}
```

forEach - inline
{println()} - lambda

Зачем нужен inline? Why is `inline` needed?

Inline нужен для уменьшения накладных расходов при использовании lambda.
Он позволяет избежать создания дополнительных объектов и вызовов функций.

Inline is used to reduce overhead when using lambdas.
It avoids creating extra objects and function calls.

Inline решает проблему:

- лишних объектов
- лишних вызовов

Inline используется когда:

- lambda вызывается часто
- важна производительность
- нужна возможность return

Пример:

`forEach`, `apply`, `run`, `let` — все inline.

Короткая суть:

- Lambda — даёт гибкость
- Inline — убирает цену за эту гибкость

Ссылка на видео этого руководства на сайте: borisproit.expert

Что такое SAM conversion? What is SAM conversion?

SAM conversion — это механизм, который позволяет передавать lambda вместо объекта интерфейса, если у него только один абстрактный метод.

Компилятор автоматически создаёт объект интерфейса из lambda.

SAM conversion allows passing a lambda instead of an object if the interface has only one abstract method.

The compiler automatically creates an interface instance from the lambda.

Здесь lambda автоматически превращается в объект `Runnable`.

```
fun runTask(task: Runnable) {
    task.run()
}

runTask {
    println("Run")   // prints Run
}
```

Что такое extension function? What is an extension function?

Ссылка на видео этого руководства на сайте: borisproit.expert

Extension function — это функция, которая добавляет новый метод к существующему классу без его изменения.

An extension function adds a new method to an existing class without modifying it.

```
fun String.lastChar(): Char {
    return this[this.length - 1]   // accessing original class
}

println("Hello".lastChar())   // using extension
```

Как работают функции с дефолтными параметрами? How do functions with default parameters work?

Функции с дефолтными параметрами позволяют не передавать аргумент, если у него уже есть значение по умолчанию.

Functions with default parameters allow skipping an argument if it already has a default value.

```
fun greet(name: String = "Guest") {
    println("Hello, $name")
}

greet()          // uses default value
greet("Alex")    // overrides default
```

C. Коллекции

Чем отличается List от MutableList? What is the difference between List and MutableList?

Ссылка на видео этого руководства на сайте: borisproit.expert

`List` — только для чтения, элементы нельзя изменять.
`MutableList` — изменяемая, можно добавлять и удалять элементы.

`List` is read-only, its elements cannot be changed.
`MutableList` is mutable, elements can be added or removed.

```
val readOnly: List<Int> = listOf(1, 2, 3)
// readOnly.add(4) ❌ not allowed

val mutable: MutableList<Int> = mutableListOf(1, 2, 3)
mutable.add(4)   // allowed
```

Чем отличается map от flatMap? What is the difference between map and flatMap?

`map` преобразует каждый элемент в один новый элемент.
`flatMap` преобразует каждый элемент в коллекцию и объединяет результат в один список.

`map` transforms each element into a single new element.
`flatMap` transforms each element into a collection and flattens the result into one list.

```
val numbers = listOf(1, 2)

println(numbers.map { listOf(it, it) })
// [[1, 1], [2, 2]]

println(numbers.flatMap { listOf(it, it) })
// [1, 1, 2, 2]
```

Что делает `filter`? What does `filter` do?

Ссылка на видео этого руководства на сайте: borisproit.expert

`filter` возвращает элементы, которые соответствуют заданному условию.

`filter` returns elements that match a given condition.

```
val numbers = listOf(1, 2, 3, 4)

val even = numbers.filter { it % 2 == 0 }   // keeps only matching elements
println(even)
```

Что такое Sequence и чем отличается от Collection? What is Sequence and how is it different from Collection?

Sequence обрабатывает элементы лениво — по одному, только когда это нужно.
Collection обрабатывает все элементы сразу.

Sequence processes elements lazily — one by one, only when needed.
Collection processes all elements eagerly.

```
val numbers = listOf(1, 2, 3, 4)

numbers.asSequence()
    .map { it * 2 }   // lazy processing
    .filter { it > 4 }
    .toList()         // triggers execution
```

Sequence нужен не для хранения, а для эффективной обработки.

Когда выбирать что?

- По умолчанию используем Collection
    - Проще
    - быстрее для маленьких списков
    - меньше накладных расходов

Ссылка на видео этого руководства на сайте: borisproit.expert

Используем Sequence когда:

- список большой
- цепочка операций длинная
- важна память
- можно остановиться раньше (например find)

Collection:

```
list
    .map { ... }
    .filter { ... }
    .map { ... }
    .filter { ... }
```

```
map → новый список
filter → новый список
map → новый список
filter → новый список
```

Sequence:

```
элемент → map → filter → результат
элемент → map → filter → результат
```

Чем fold отличается от reduce? What is the difference between fold and reduce?

`reduce` использует первый элемент как начальное значение.
`fold` позволяет задать начальное значение вручную.

Ссылка на видео этого руководства на сайте: borisproit.expert

---

`reduce` uses the first element as the initial value.
`fold` allows providing an explicit initial value.

```
val numbers = listOf(1, 2, 3)

println(numbers.reduce { acc, i -> acc + i })   // prints 6

println(numbers.fold(10) { acc, i -> acc + i }) // prints 16
```

D. Scope functions

Что делают let / run / apply / also / with? What do let / run / apply / also / with do?

Это scope functions — они позволяют работать с объектом внутри блока кода.
Отличаются тем, как передают объект и что возвращают.

These are scope functions — they allow working with an object inside a block.
They differ in how they pass the object and what they return.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
val text = "Hello"

text.let {
    println(it)     // prints Hello
}

text.run {
    println(length) // prints 5
}

val builder = StringBuilder().apply {
    append("Hi")
}
println(builder)    // prints Hi

text.also {
    println(it)     // prints Hello
}

with(text) {
    println(length) // prints 5
}
```

`let` — когда объект может быть `null` и/или нужно получить результат

Самый частый кейс: **nullable**

```
val length = name?.let { it.length }  // если name != null → вернёт length
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Или когда надо сделать несколько действий и вернуть **что-то новое**:

```
val result = user.let {
    println(it.id)
    it.name.uppercase()
}
```

`run` = Часто используют для “локального блока”

как `let`, но внутри `this` (удобно, когда много обращений к полям)

```
val result = user.run {
    println(id)
    name.uppercase()
}
```

```
val url = run {
    val host = "example.com"
    "https://$host/api"
}
```

`apply` — настройка объекта (builder-стиль). Всегда возвращает объект. “настроил и вернул объект”.

```
val paint = Paint().apply {
    isAntiAlias = true
    strokeWidth = 4f
}
```

Ссылка на видео этого руководства на сайте: borisproit.expert

`also` — побочные действия (лог, аналитика), но не менять цепочку. Возвращает объект. “сделал что-то рядом, но вернул исходный объект”.

```
val result = data
    .also { println("Loaded: $it") }
    .also { analytics.log("loaded") }
```

`with(obj)` — как `run`, но объект передаётся параметром. Используют, когда объект уже есть и хочется “поработать внутри `this`” блоком.

```
val result = with(user) {
    println(id)
    name.uppercase()
}
```

Как выбрать?
Нужно обработать nullable / вернуть значение → `let`
Нужно вычислить результат, много обращений к полям → `run` или `with`
Нужно сконфигурировать объект → `apply`
Нужно логирование/аналитика/побочные эффекты, не ломая цепочку → `also`

Когда использовать apply vs let? When to use apply vs let?

`apply` используют для настройки объекта.
`let` используют для работы с результатом или null-проверки.

`apply` is used for configuring an object.
`let` is used for working with a result or null-check.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
val user = StringBuilder().apply {
    append("Alex")
}
println(user)   // prints Alex

val name: String? = "John"
name?.let {
    println(it.length)   // prints 4
}
```

Почему apply возвращает объект, а let — результат?
Why does apply return the object while let returns the result?

`apply` предназначен для настройки объекта, поэтому возвращает сам объект.
`let` предназначен для работы с результатом, поэтому возвращает результат блока.

`apply` is meant for configuring an object, so it returns the object itself.
`let` is meant for working with a result, so it returns the lambda result.

```
val builder = StringBuilder().apply {
    append("Hi")
}
println(builder)   // prints Hi

val length = "Hello".let {
    it.length
}
println(length)   // prints 5
```

Ссылка на видео этого руководства на сайте: borisproit.expert

E. Иммутабельность

Почему immutable объекты безопаснее? Why are immutable objects safer?

Immutable объекты нельзя изменить после создания, поэтому их состояние всегда предсказуемо.

Immutable objects cannot be changed after creation, so their state is always predictable.

```
val numbers = listOf(1, 2, 3)

// numbers.add(4) ❌ not allowed

println(numbers)   // prints [1, 2, 3]
```

Почему shared mutable state — опасен? Why is shared mutable state dangerous?

Общий изменяемый объект может быть изменён разными потоками, что приводит к непредсказуемому состоянию.

Shared mutable state can be modified by multiple threads, leading to unpredictable state.

G. Делегирование

Что такое delegation? What is delegation?

Delegation — это передача ответственности другому объекту вместо самостоятельной реализации.

Delegation is passing responsibility to another object instead of implementing it yourself.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
private val binding by viewBinding(ActivityMainBinding::inflate)
```

Мы делегируем создание binding

```
private val viewModel: MainViewModel by viewModels()
```

Ты не создаёшь ViewModel сам.

- Делегат делает это за тебя
- с правильным lifecycle

Что такое delegated properties? What are delegated properties?

Delegated properties — это свойства, чья логика хранения или вычисления передаётся другому объекту.

Delegated properties are properties whose logic is delegated to another object.

Вместо:

```
var token = ""
```

где значение хранится в памяти,

мы делаем:

```
var token by preferenceDelegate
```

Теперь значение:

- может храниться в SharedPreferences
- в Bundle
- в файле
- где угодно

Ссылка на видео этого руководства на сайте: borisproit.expert

Что такое `by lazy`? What is `by lazy`?

`by lazy` — это делегат, который откладывает инициализацию свойства до первого обращения.

`by lazy` is a delegate that delays property initialization until the first access.

Создай значение только тогда, когда оно впервые понадобится

```
val repository by lazy {
    UserRepository()
}
```

Или

```
val repository = UserRepository()
```

```
val viewModel by lazy {
    ViewModelProvider(this)[MainViewModel::class.java]
}
```

создаётся только если экран реально использует его

Чем отличается `lateinit` от `lazy`?
What is the difference between `lateinit` and `lazy`?

`lateinit` — отложенная инициализация переменной, которую нужно присвоить вручную.
`lazy` — автоматическая инициализация при первом обращении.

`lateinit` is deferred initialization that must be assigned manually.
`lazy` initializes automatically on first access.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
lateinit var name: String

val title: String by lazy {
    "Hello"
}

name = "Alex"

println(name)   // prints Alex
println(title)  // prints Hello
```

Что такое interface delegation (by)? What is interface delegation (by)?

Interface delegation позволяет передать реализацию интерфейса другому объекту.

Interface delegation allows delegating interface implementation to another object.

H. Модификаторы

Что делают private / internal / protected / public? What do private / internal / protected / public do?

Это модификаторы доступа, которые определяют, где можно использовать класс, функцию или свойство.

These are access modifiers that define where a class, function, or property can be used.

Private - Видно только внутри класса (или файла)

Protected - Видно внутри класса и его наследников

Ссылка на видео этого руководства на сайте: borisproit.expert

Internal - Видно внутри модуля

public (по умолчанию) - Видно везде

Что такое `open`? What is `open`?

`open` — это ключевое слово, которое разрешает наследование класса или переопределение функции.

`open` is a keyword that allows a class to be inherited or a function to be overridden.

В Kotlin всё закрыто по умолчанию

Вот так:

```
class Animal
```

этот класс **нельзя наследовать**

```
class Dog : Animal() // ❌ ошибка
```

Чтобы разрешить наследование

Нужно написать:

```
open class Animal
```

Теперь:

```
class Dog : Animal() // ✅ работает
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Почему классы final по умолчанию? Why are classes final by default?

Классы final по умолчанию, чтобы предотвратить случайное наследование и сохранить предсказуемое поведение.

Classes are final by default to prevent accidental inheritance and keep behavior predictable.

I. Equals / HashCode

Как data class генерирует equals? How does a data class generate equals?

Data class генерирует equals на основе всех свойств, указанных в конструкторе.

A data class generates equals based on all properties defined in the primary constructor.

```
data class User(val name: String, val age: Int)

val u1 = User("Alex", 25)
val u2 = User("Alex", 25)

println(u1 == u2)   // prints true
```

Когда нужно override equals/hashCode вручную? When should equals/hashCode be overridden manually?

Когда логика равенства зависит не от всех свойств объекта.

When equality logic depends on only some of the object's properties.

Ссылка на видео этого руководства на сайте: borisproit.expert

```kotlin
class User(val id: Int, val name: String) {

    override fun equals(other: Any?): Boolean {
        return other is User && other.id == id
    }

    override fun hashCode(): Int {
        return id
    }
}

val u1 = User(1, "Alex")
val u2 = User(1, "John")

println(u1 == u2)   // prints true
```

J. Sealed thinking

Почему sealed class лучше enum для state? Why is sealed class better than enum for state?

`sealed class` лучше для state, потому что он позволяет описывать разные состояния с разными данными (например, `Success(data)` или `Error(message)`), и Kotlin заставляет `when` быть исчерпывающим (ты не забудешь обработать состояние). `enum` — это просто фиксированный список значений без нормальной типобезопасной “нагрузки” данными.

`sealed class` is better for state because it can model states with different payloads (e.g., `Success(data)` or `Error(message)`) and Kotlin can enforce exhaustive `when` handling. `enum` is just a fixed set of constants and doesn’t naturally model typed data per state.

Ссылка на видео этого руководства на сайте: borisproit.expert

```kotlin
// ✅ Sealed: each state can have its own data
sealed class UiState {
    data object Loading : UiState() // Represents loading state
    data class Success(val items: List<String>) : UiState() // Holds result data
    data class Error(val message: String, val code: Int? = null) : UiState() // Holds error detail
}

fun render(state: UiState) {
    when (state) {
        UiState.Loading -> println("Loading...")
        is UiState.Success -> println("Items: ${state.items}")
        is UiState.Error -> println("Error: ${state.message}, code=${state.code}")
    }
}

// ❌ Enum: no per-state typed payload
enum class UiStateEnum { Loading, Success, Error }

// Where do we store items/message?
// Usually in separate variables,
// which creates risk of inconsistency (e.g. state = Success, but data = null)
```

Как sealed class помогает архитектуре? How does a sealed class help architecture?

Sealed class позволяет описать все возможные состояния системы явно и безопасно.

A sealed class allows defining all possible system states explicitly and safely.

Ссылка на видео этого руководства на сайте: borisproit.expert

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

fun handle(result: Result) {
    when(result) {
        is Result.Loading -> println("Loading")
        is Result.Success -> println(result.data)
        is Result.Error -> println(result.message)
    }
}

handle(Result.Success("OK"))   // prints OK
```

K. Inline deeper

Что такое `crossinline`? What is `crossinline`?

`crossinline` запрещает нелокальный return из lambda внутри inline функции.

`crossinline` prevents non-local returns from a lambda inside an inline function.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
inline fun runTask(block: () -> Unit) {
    block()
}

fun test() {
    runTask {
        return   // ← выходит из test()
    }

    println("This will NOT run")
}
```

он завершает всю функцию `test()`
`lambda → return → test() заканчивается`

```
inline fun runTask(crossinline block: () -> Unit) {
    block()
}

fun test() {
    runTask {
        return   // ❌ нельзя выйти из test()
    }
}
```

Ссылка на видео этого руководства на сайте: borisproit.expert

```
fun test() {
    runTask {
        return@runTask   // ← завершает только lambda
    }

    println("This WILL run")
}
```

`lambda → return@runTask → только lambda закончилась`

Что такое `noinline`? What is `noinline`?

`noinline` — это модификатор для параметра-lambda внутри `inline` функции, который говорит компилятору: **“эту лямбду не подставляй в место вызова, оставь как объект”**.
То есть `inline` обычно старается “развернуть” лямбду прямо в код, чтобы не создавать лишний объект, а `noinline` специально **запрещает** это для конкретной лямбды.

Зачем это нужно на практике:

1. **Если лямбду нужно сохранить в переменную или передать дальше как значение**
   Инлайн-лямбду нельзя нормально “положить в переменную”, потому что её логика должна быть вшита в место вызова.

2. **Если лямбду нужно передать в не-inline API**
   Например, в функцию/объект, который принимает обычный `() -> Unit` и будет вызывать его позже.

`noinline` is a modifier for a lambda parameter inside an `inline` function that tells the compiler: **“do not inline this lambda, keep it as an object”**.
Normally, `inline` tries to inline lambdas to avoid allocations, but `noinline` explicitly prevents that for a specific lambda.

Why it is useful:

Ссылка на видео этого руководства на сайте: borisproit.expert

1. **When you need to store a lambda in a variable or return it**
   Inlined lambdas are not meant to exist as objects.

2. **When you need to pass it to a non-inline API**
   For example, to something that will call it later.

```
inline fun runBoth(
    action: () -> Unit,
    noinline callback: () -> Unit
) {
    action()   // inlined

    val savedCallback = callback   // allowed because callback is not inlined
    savedCallback()                // calls the stored lambda
}

runBoth(
    { println("Action") },    // prints Action
    { println("Callback") }   // prints Callback
)
```

L. Object lifecycle

Когда создаётся `object`? When is an `object` created?

`object` создаётся при первом обращении к нему (ленивая инициализация). Он НЕ создаётся при запуске приложения.

An `object` is created on first access (lazy initialization).

Ссылка на видео этого руководства на сайте: borisproit.expert

```
object Logger {
    init {
        println("Logger created")   // prints when first accessed
    }
}

println("Start")
Logger   // first access → creates object
```

Чем `object` отличается от `companion object`? What is the difference between `object` and `companion object`?

`object` — это самостоятельный singleton.
`companion object` — это singleton внутри класса для статического доступа.

```
object Logger {
    fun log() = println("Logging")   // prints Logging
}

class User {
    companion object {
        fun create() = User()
    }
}

Logger.log()        // prints Logging
User.create()       // creates instance
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Что делает `init` блок? What does the `init` block do?

`init` выполняется при создании объекта и используется для инициализации.

`init` runs when the object is created and is used for initialization.

```
class User(name: String) {

    init {
        println("User created: $name")   // prints User created: Alex
    }
}

val user = User("Alex")
```

Чем inner class отличается от nested class? What is the difference between inner class and nested class?

`nested class` не имеет доступа к внешнему классу.
`inner class` имеет доступ к внешнему классу.

A `nested class` has no access to the outer class.
An `inner class` has access to the outer class.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
class Outer(val name: String) {

    class Nested {
        fun print() = println("No access")   // prints No access
    }

    inner class Inner {
        fun print() = println(name)   // prints Alex
    }
}

Outer.Nested().print()
Outer("Alex").Inner().print()
```

Может ли быть два экземпляра `object`? Can there be two instances of an `object`?

Нет, `object` создаёт только один экземпляр (singleton).

No, an `object` creates only one instance (singleton).

```
object Logger

val a = Logger
val b = Logger

println(a === b)   // prints true
```

Ссылка на видео этого руководства на сайте: borisproit.expert

M. Functional thinking

Что такое immutability-first подход? What is the immutability-first approach?

Immutability-first — это подход, при котором объекты по умолчанию создаются неизменяемыми.

- Упрощает понимание кода — состояние не меняется неожиданно
- Уменьшает количество багов
- Делает код безопаснее в многопоточности
- Обеспечивает предсказуемое поведение состояния
- Упрощает тестирование
- Хорошо подходит для архитектур с управлением состоянием (например, MVI)
- Упрощает отслеживание изменений между версиями данных

Immutability-first is an approach where objects are immutable by default.

```
data class User(val name: String, val age: Int)

val user = User("Alex", 25)
// user.age = 26 ❌ not allowed

println(user)   // prints User(name=Alex, age=25)
```

N. Variance

Что такое `out` и `in`? What are `out` and `in`?

`out` — позволяет только получать значения (producer).
`in` — позволяет только передавать значения (consumer).

Ссылка на видео этого руководства на сайте: borisproit.expert

`out` allows only producing values (read-only).
`in` allows only consuming values (write-only).

```
interface Box<out T> {
    fun get(): T
}
```

T используется только как возвращаемый тип.

```
interface Consumer<in T> {
    fun consume(value: T)
}
```

T используется только как входной параметр.

O. Kotlin vs Java

Что использовать в Kotlin вместо static? What to use in Kotlin instead of static?

В Kotlin вместо static используют `companion object` или `object`.

In Kotlin, `companion object` or `object` are used instead of static.

```
class Utils {
    companion object {
        fun greet() = println("Hello")   // prints Hello
    }
}

Utils.greet()
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Чем Any отличается от Object? What is the difference between Any and Object?

`Any` — это базовый тип в Kotlin, не связанный с JVM.
`Object` — базовый класс в Java.

`Any` is the root type in Kotlin, not tied to JVM implementation.
`Object` is the root class in Java.

Чем отличается List от Array?
What is the difference between List and Array?

`Array` — это базовый контейнер фиксированного размера, который хранит элементы напрямую.
`List` — это более высокий уровень абстракции поверх коллекций с функциональными операциями.

Ключевые отличия:

- `Array` — часть базовой модели языка, ближе к JVM-массивам
- `List` — интерфейс коллекции из стандартной библиотеки
- `Array` изменяем по элементам
- `List` по умолчанию read-only (но есть MutableList)
- `List` поддерживает map, filter, flatMap и т.д.
- `Array` требует конвертации для функциональной обработки

`Array` is a low-level fixed-size container storing elements directly.
`List` is a higher-level abstraction with functional operations.

Key differences:

- `Array` is part of the core language model
- `List` is a collection interface from the standard library
- `Array` is mutable by elements
- `List` is read-only by default (MutableList exists)
- `List` supports map, filter, flatMap etc.
- `Array` usually needs conversion for functional processing

Ссылка на видео этого руководства на сайте: borisproit.expert

```
val array = arrayOf(1, 2, 3)
array[0] = 10

val list = listOf(1, 2, 3)
// list[0] = 10 ❌ not allowed

val result = array.toList().map { it * 2 }
println(result)   // prints [20, 4, 6]
```