Ссылка на видео этого руководства на сайте: borisproit.expert

🟢 ОБЯЗАТЕЛЬНО ЗНАТЬ (Junior → Strong Junior → Middle base)

A. Базовые концепции

1. Что такое корутины в Kotlin? What are coroutines in Kotlin?

Kotlin-корутины — это функциональность на уровне языка, которая позволяет писать асинхронный и неблокирующий код в привычном, последовательном стиле.
Механизм корутин встроен в компилятор Kotlin, а библиотека kotlinx.coroutines предоставляет готовые структурированные API, такие как launch, async, Flow и dispatchers.

Kotlin coroutines are a language-level feature that allows writing asynchronous, non-blocking code in a sequential style. The coroutine mechanism is built into the Kotlin compiler, and the kotlinx.coroutines library provides structured APIs such as launch, async, Flow and dispatchers.

```
viewModelScope.launch {
    val data = repository.loadData() // suspend function
    _state.value = data
}
```

2. Чем корутина отличается от потока? How is a coroutine different from a thread?

Корутина — это лёгкая единица работы, которая выполняется внутри потока. Она может приостанавливаться и возобновляться, не блокируя поток, и при этом занимает очень мало ресурсов. В одном потоке могут выполняться тысячи корутин одновременно. Поток же — это тяжёлый системный объект операционной системы, создание и переключение между потоками требует заметных ресурсов, и их количество ограничено.

A coroutine is a lightweight unit of work that runs inside a thread. It can be suspended and resumed without blocking the thread, while consuming very few resources. Thousands of coroutines can run within a single thread. A thread, on the other hand, is a heavyweight operating system resource, and creating or switching between threads is relatively expensive and limited.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Thread
Thread {
    Thread.sleep(1000) // blocks thread
}.start()

// Coroutine
viewModelScope.launch {
    delay(1000) // does not block thread
}
```

3. **Что такое suspend-функция? What is a suspend function?**

suspend-функция — это функция, которая может быть приостановлена во время выполнения и возобновлена позже без блокировки потока. Она используется внутри корутин и позволяет выполнять длительные операции (например, сетевые запросы или работу с базой данных), не замораживая основной поток приложения. Вызов suspend-функции возможен только из корутины или другой suspend-функции.

A suspend function is a function that can be paused during execution and resumed later without blocking the thread. It is used inside coroutines and allows performing long-running operations, such as network requests or database access, without freezing the main thread. A suspend function can only be called from a coroutine or another suspend function.

```
suspend fun loadData(): String {
    delay(1000) // suspends coroutine, not thread
    return "Result"
}

viewModelScope.launch {
    val result = loadData()
    println(result)
}
```

Ссылка на видео этого руководства на сайте: borisproit.expert

4. Что такое CoroutineScope и зачем он нужен?What is CoroutineScope and why is it needed?

CoroutineScope — это объект, который определяет границы жизни корутин. Все корутины, запущенные внутри этого scope, связаны с ним и автоматически отменяются при его завершении. Это позволяет безопасно управлять асинхронной работой и предотвращать утечки памяти. Например, у ViewModel есть свой scope, и корутины в нём живут до тех пор, пока жива ViewModel.

CoroutineScope is an object that defines the lifetime boundaries for coroutines. All coroutines launched within this scope are tied to it and are automatically cancelled when the scope is cancelled or completed. This provides safe lifecycle management for asynchronous work and helps prevent memory leaks. For example, a ViewModel has its own scope, and coroutines inside it live as long as the ViewModel does.

```
viewModelScope.launch {
    loadData()
}
```

5. Что такое Job?What is a Job?

Job — это объект, представляющий выполняющуюся корутину. Он хранит её состояние (активна, завершена, отменена) и позволяет управлять её жизненным циклом: например, отменять выполнение или отслеживать завершение. Job также может иметь дочерние задачи, формируя иерархию — при отмене родительского Job автоматически отменяются все дочерние корутины.

A Job is an object that represents a running coroutine. It stores the coroutine's state (active, completed, or cancelled) and allows managing its lifecycle, such as cancelling it or waiting for it to finish. A Job can also have child jobs, forming a hierarchy where cancelling a parent Job automatically cancels all of its child coroutines.

```
viewModelScope.launch {   // ← Parent Job

    launch {              // ← Child 1
        loadUserProfile()
    }

    launch {              // ← Child 2
        loadBookings()
    }
}
```

Ссылка на видео этого руководства на сайте: borisproit.expert

6. Что такое CoroutineContext? What is CoroutineContext?

CoroutineContext — это набор настроек для корутины, который обычно включает:

1. Job — управляет жизненным циклом корутины (lifecycle)
варианты: Job, SupervisorJob

2. Dispatcher — определяет, где выполняется корутина (thread / thread-pool)
варианты: Dispatchers.Main, Dispatchers.IO, Dispatchers.Default, Dispatchers.Unconfined

3. CoroutineName — имя корутины (для дебага и логов)
варианты: CoroutineName("LoadGuest"), CoroutineName("SyncPrices")

4. CoroutineExceptionHandler — обработка ошибок (global error handling)
варианты: CoroutineExceptionHandler { context, exception -> ... }

5. Пользовательские элементы — дополнительные данные, пробрасываемые через контекст
варианты: кастомные классы, реализующие CoroutineContext.Element

И комбинация этих настроек определяет,
как именно будет жить, выполняться и обрабатываться корутина.

CoroutineContext is a set of configuration elements that define how a coroutine runs. It usually includes:

1. Job — controls the coroutine lifecycle
options: Job, SupervisorJob

2. Dispatcher — defines the execution thread or thread pool
options: Dispatchers.Main, Dispatchers.IO, Dispatchers.Default, Dispatchers.Unconfined

3. CoroutineName — gives the coroutine a readable name for debugging/logging
options: CoroutineName("LoadGuest"), CoroutineName("SyncPrices")

4. CoroutineExceptionHandler — handles uncaught coroutine exceptions
options: CoroutineExceptionHandler { context, exception -> ... }

5. Custom context elements — user-defined metadata passed through context
options: any class implementing CoroutineContext.Element

Ссылка на видео этого руководства на сайте: borisproit.expert

And the combination of these elements determines
how the coroutine is created, executed, cancelled, resumed, and logged.

```
private val exceptionHandler = CoroutineExceptionHandler { _, throwable ->
    logger.log("PriceSync failed: ${throwable.message}")
}

private val scope = CoroutineScope(
    SupervisorJob() +
    Dispatchers.IO +
    CoroutineName("PriceSyncScope") +
    exceptionHandler
)

fun startSync() {
    scope.launch(CoroutineName("SyncPrices")) {
        val prices = api.fetchLatestPrices()
        dao.savePrices(prices)
    }
}

fun stop() {
    scope.cancel()
}
```

6. Что такое Dispatcher? What is a Dispatcher?

Dispatcher — это часть CoroutineContext, которая определяет,
на каком потоке или пуле потоков будет выполняться корутина.
Он управляет тем, где именно выполняется код — на UI-потоке, в фоне или в CPU-пуле.

В Kotlin есть стандартные диспетчеры:

1. Dispatchers.Main — главный (UI) поток.
Используется для работы с интерфейсом.

2. Dispatchers.IO — пул потоков для операций ввода-вывода.
Сеть, база данных, файлы.

Ссылка на видео этого руководства на сайте: borisproit.expert

3. Dispatchers.Default — пул потоков для CPU-нагрузки.
Работа с коллекциями, вычисления, парсинг.

4. Dispatchers.Unconfined — спец-режим.
Корутину не привязывает к какому-то потоку. Обычно не используется в продакшене.

Dispatcher отвечает за:

- выбор потока выполнения
- переключение контекста между потоками
- оптимальное распределение задач

A Dispatcher is a part of the CoroutineContext that defines
which thread or thread pool a coroutine runs on.
It controls whether the coroutine executes on the UI thread, a background IO pool, or a CPU-optimized pool.

Kotlin provides standard dispatchers:

1. Dispatchers.Main — main/UI thread
Used for UI updates.

2. Dispatchers.IO — IO-optimized thread pool
Used for networking, DB, and file operations.

3. Dispatchers.Default — CPU-optimized pool
Used for heavy computation and data processing.

4. Dispatchers.Unconfined — executes without being confined to a specific thread
Mostly for advanced use/debugging.

The dispatcher schedules coroutine execution and handles switching between threads when needed.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// UI – update screen
GlobalScope.launch(Dispatchers.Main) {
    println("Main: UI thread")

    // IO – network call
    val rooms = withContext(Dispatchers.IO) {
        println("IO: loading rooms from API")
        listOf("Room A", "Room B")
    }

    // CPU – process result
    val formatted = withContext(Dispatchers.Default) {
        println("Default: processing data")
        rooms.joinToString()
    }

    // Unconfined – advanced use/debugging
    withContext(Dispatchers.Unconfined) {
        println("Unconfined: continues wherever resumed")
    }
```

7. Что такое structured concurrency? What is structured concurrency?

Structured concurrency — это принцип работы корутин, при котором каждая корутина имеет владельца и принадлежит определённому scope, а её жизненный цикл контролируется родителем.

Главные идеи:

1. У каждой корутины есть scope-контейнер (например viewModelScope, lifecycleScope, coroutineScope {})

2. Родитель ждёт завершения всех дочерних корутин корутины не “теряются” в фоне

3. Ошибки и отмена распространяются по иерархии родитель → дети

4. Нет “висящих” фоновых задач
→ меньше утечек памяти
→ чище архитектура
→ контролируемый lifecycle

Ссылка на видео этого руководства на сайте: borisproit.expert

Простыми словами:
корутины живут строго внутри структурированной иерархии,
и система гарантирует, что они корректно завершатся.

Structured concurrency is a design principle where
every coroutine has a well-defined parent scope and lifecycle,
and its execution is managed by that parent.

Key ideas:

1. Every coroutine belongs to a scope
(e.g. viewModelScope, lifecycleScope, coroutineScope {})

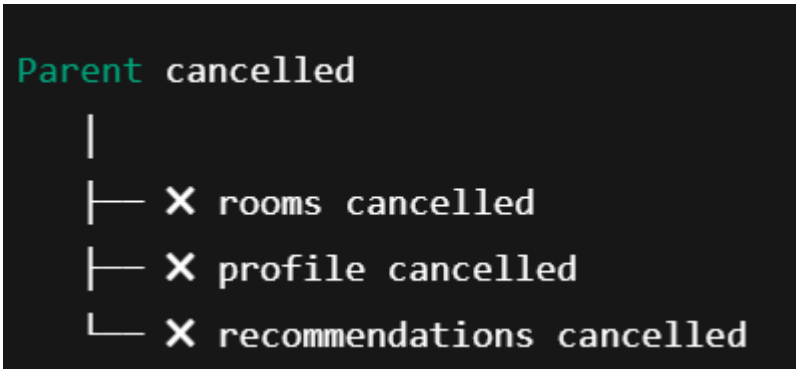
2. The parent waits for all child coroutines to finish
so no background tasks are “lost”

3. Cancellation and errors propagate through the hierarchy
parent → children

4. No uncontrolled background work
→ fewer leaks
→ predictable lifecycle
→ safer async code

In short:
coroutines run inside a structured tree of scopes instead of being fire-and-forget.

Ссылка на видео этого руководства на сайте: borisproit.expert



B. Создание и запуск корутин

11. Чем launch отличается от async? What is the difference between launch and async?

launch — это «сделай что-то».
Корутину запускаем, она делает работу (в том числе может вызвать API и получить результат),
но сама корутина результат наружу не возвращает.
Мы просто выполняем действие.

Ссылка на видео этого руководства на сайте: borisproit.expert

async — это «посчитай и верни значение».
Корутину запускаем, она что-то считает (например, делает API-запрос)
и мы ждём от неё результат через await().

Главная мысль:
launch — для действий
async — для получения результата

И да — API-вызов может возвращать данные в обоих случаях,
разница только в том, нужен ли этот результат «снаружи корутины».

---

launch means “do something”.
The coroutine runs and may even call an API and get data,
but the coroutine itself does not return a value outside.
It is used for actions.

async means “calculate and return a value”.
The coroutine runs (for example, calls an API)
and we expect a result from it using await().

Core idea:
launch — do an action
async — get a result

And yes — an API call may return data in both cases —
the difference is whether we need that data *outside* the coroutine.

```
viewModelScope.launch {
    val room = api.getRoomDetails(roomId) // suspend fun
    uiState.value = room
}
```

Ссылка на видео этого руководства на сайте: borisproit.expert

```
viewModelScope.launch {

    val image = async { api.getRoomImage(roomId) }
    val description = async { api.getRoomDescription(roomId) }

    val room = RoomUi(
        image = image.await(),
        description = description.await()
    )

    uiState.value = room
}
```

12. Что возвращает launch и что возвращает async? What does launch return and what does async return?

launch возвращает Job — это «дескриптор корутины».
Через него можно проверить состояние корутины или отменить её.
Но значение корутина не возвращает.

async возвращает Deferred<T> —
это тот же Job, только с результатом типа T внутри.
Таким типом может быть любая твоя модель, например:

Deferred<RoomInfo>

Deferred<List<Booking>>

Deferred<GuestProfile>

Когда ты вызываешь await(),
ты получаешь обычный объект нужного типа.

Главная мысль:
Job — просто задача
Deferred<T> — задача с результатом

---

launch returns a Job — a handle to a running coroutine.
You can cancel it or check its state,
but it does not return a value.

Ссылка на видео этого руководства на сайте: borisproit.expert

async returns a Deferred<T> —
which is like a Job, but it holds a result of type T.
That T may be any model, for example:

Deferred<RoomInfo>

Deferred<List<Booking>>

Deferred<GuestProfile>

When you call await(),
you get the actual object of that type.

Core idea:
Job — task only
Deferred<T> — task + result

```
val job: Job = viewModelScope.launch {
    doWork()
}

val deferred: Deferred<Int> = viewModelScope.async {
    loadNumber()
}

val result = deferred.await()
```

13. Что делает runBlocking и где его можно использовать? What does runBlocking do and why is it used?

runBlocking — это функция, которая запускает корутину и при этом блокирует текущий поток до тех пор, пока корутина не завершится.
То есть корутина работает, а поток просто стоит и ждёт её окончания — и только потом код продолжается.

Это полезно там, где результат нужен прямо сейчас и можно позволить себе блокировку:

Ссылка на видео этого руководства на сайте: borisproit.expert

- в юнит-тестах
- в консольных программах
- иногда при миграции старого синхронного кода

Но в Android-UI runBlocking использовать нельзя, потому что он «замораживает» поток и может вызвать подвисание приложения.

Главная идея:
runBlocking — это способ дождаться корутины, заблокировав поток.

---

runBlocking is a function that starts a coroutine and blocks the current thread until that coroutine finishes.
So the coroutine runs, while the thread simply waits and does nothing else — only after the coroutine completes does execution continue.

It is useful when you need the result right now and blocking is acceptable, such as:

- in unit tests
- in console applications
- sometimes when migrating legacy synchronous code

But it must not be used in Android UI code, because blocking the main thread can freeze the app.

Core idea:
runBlocking is a way to wait for a coroutine by blocking the thread.

```
fun main() = runBlocking {
    val result = loadData()
    println(result)
}
```

14. Чем runBlocking отличается от coroutineScope? What is the difference between runBlocking and coroutineScope?

Главное отличие — в поведении потока. runBlocking запускает корутину и блокирует текущий поток, пока она не завершится. Поток просто ждёт и не делает больше ничего.

Ссылка на видео этого руководства на сайте: borisproit.expert

coroutineScope запускает корутины и приостанавливает текущую корутину, но поток НЕ блокируется. То есть другие задачи на этом потоке могут продолжать выполняться.

The key difference is in how they treat the thread.

runBlocking
starts a coroutine and blocks the current thread until it finishes.
The thread simply waits and does nothing else.

coroutineScope
starts coroutines and suspends the current coroutine, but does NOT block the thread.
So other work on the same thread can continue executing.

```
// runBlocking – thread IS blocked
fun main() = runBlocking {
    val room = api.getRoom(roomId)
    println(room.name)
}

// coroutineScope – thread is NOT blocked
suspend fun loadRoom() = coroutineScope {
    val room = api.getRoom(roomId)
    println(room.name)
}
```

15. Что делает withContext? What does withContext do?

withContext — это функция, которая переключает корутину в другой контекст (обычно — другой Dispatcher), при этом:

- текущая корутина приостанавливается
- выполнение продолжается в другом потоке/диспетчере

Ссылка на видео этого руководства на сайте: borisproit.expert

- после завершения — корутина возвращается обратно в исходный контекст

При этом поток не блокируется — меняется только контекст выполнения.

Чаще всего withContext используют для:

- Dispatchers.IO — сеть, файлы, база данных
- Dispatchers.Default — тяжёлые вычисления
- Dispatchers.Main — обновление UI

Главная идея:

withContext = временно выполнить код в другом месте

---

withContext is a function that switches the coroutine to a different context (usually another Dispatcher):

- the current coroutine is suspended
- execution continues in the new dispatcher/thread
- when finished, it returns to the original context

The thread is NOT blocked — only the coroutine context changes.

Typical usage:

- Dispatchers.IO — networking, files, database
- Dispatchers.Default — CPU-heavy work
- Dispatchers.Main — UI updates

Ссылка на видео этого руководства на сайте: borisproit.expert

```
viewModelScope.launch(Dispatchers.Main) {

    // switch to IO for API request
    val room = withContext(Dispatchers.IO) {
        api.getRoomDetails(roomId)
    }

    // back on Main — safe to update UI
    uiState.value = room
}
```

C. Отмена и lifecycle

17. Как отменить корутину? How to cancel coroutine?

Корутину отменяют через её Job: при запуске корутины сохраняют job, затем вызывают job.cancel(). Отмена кооперативная — корутина завершится в ближайшей точке приостановки.

A coroutine is cancelled via its Job: when launching it, keep the returned job, then call job.cancel(). Cancellation is cooperative — the coroutine stops at the next suspension point.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
val job = viewModelScope.launch {
    loadData() // suspend fun
}

job.cancel() // cancel coroutine
```

18. Что такое кооперативная отмена? What is cooperative cancellation?

Кооперативная отмена — это когда корутина не прерывается «силой», а сама корректно завершается, когда замечает, что её отменили.
Она делает это:

- в точках приостановки
(delay, await, withContext, I/O и т.д.)
- во время suspend API-вызовов (например Retrofit)
- или при ручной проверке isActive

То есть выполнение просто не продолжается дальше.

Cooperative cancellation means that a coroutine is not force-stopped.
Instead, it terminates gracefully when it detects cancellation:

- at suspension points
(delay, await, withContext, I/O, etc.)
- during suspend API calls (e.g., Retrofit)
- or when explicitly checking isActive

Execution simply does not continue after cancellation.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
val job = viewModelScope.launch {

    // suspend API call – cancellation-aware
    val room = api.getRoomDetails(roomId)

    println(room.name) // runs only if not cancelled
}

job.cancel() // cancels coroutine + API call
```

19. Что такое isActive и зачем он нужен?

isActive — это флаг внутри корутины, который показывает, не была ли она отменена.
Его используют, чтобы вручную останавливать длительные операции (например, циклы или вычисления), если корутину отменили.

---

isActive is a flag available inside a coroutine that indicates whether it is still active and not cancelled.
It is used to manually stop long-running work (like loops or calculations) when the coroutine gets cancelled.

```
val job = viewModelScope.launch {

    while (isActive) {     // stop when cancelled
        val rooms = api.getRooms()
        delay(1000)
    }
}

job.cancel() // coroutine stops on next check
```

Ссылка на видео этого руководства на сайте: borisproit.expert

20. Как отмена распространяется на дочерние корутины?

В корутинах действует иерархия: если отменяется родительская корутина (или scope), то автоматически отменяются все её дочерние корутины.
Это и есть часть *structured concurrency* — дети не могут жить дольше родителя.

Coroutines form a parent–child hierarchy.
When the parent coroutine (or scope) is cancelled, all of its child coroutines are cancelled automatically.
This is part of structured concurrency — children cannot outlive the parent.

```
val job = viewModelScope.launch {

    launch { api.syncRooms() }      // child 1
    launch { api.syncBookings() }   // child 2
}

job.cancel() // cancels parent + both children
```

D. Ошибки и исключения

22. Как обрабатывать исключения внутри корутины? How do you handle exceptions inside a coroutine?

Исключения в корутинах обрабатываются внутри самой корутины через try/catch — то есть внутри launch { ... } или async { ... }, а не снаружи.
Отмена корутины (CancellationException) — это нормальное поведение, оно не крашит приложение, и обычно её не “глотают”, а пробрасывают дальше.
А вот реальные ошибки (сети, БД, логики) ловят в catch и показывают ошибку пользователю или логируют.

Exceptions in coroutines are handled inside the coroutine body using try/catch, i.e. inside launch { ... } or async { ... }, not around them.
Coroutine cancellation (CancellationException) is normal behavior and does not crash the app; it is

usually rethrown instead of being swallowed.
Real errors (network, DB, logic) are caught in catch and used to show an error or log it.

```
viewModelScope.launch {
    try {
        val room = api.getRoomDetails(roomId) // suspend API call
        uiState.value = room
    } catch (e: CancellationException) {
        throw e             // keep normal coroutine cancellation
    } catch (e: Exception) {
        showError(e.message ?: "Unknown error")
    }
}
```

23. **Чем отличается try/catch от CoroutineExceptionHandler? What is the difference between `try / catch` and `CoroutineExceptionHandler`?**

try/catch — используется внутри корутины
и ловит ошибки локально там, где они происходят.
Это бизнес-логика обработки ошибок.

CoroutineExceptionHandler — ловит
неперехваченные исключения верхнего уровня
(например, из launch),
и нужен как «глобальная страховка» приложения
для логирования, crash-репортинга, fallback-поведения.

Лучше использовать оба:
try/catch — локально
ExceptionHandler — глобально.

---

try/catch handles exceptions inside a coroutine,
locally where the error happens.
It is used for business-logic error handling.

CoroutineExceptionHandler catches
uncaught coroutine exceptions at the top level
(for example, from launch),
and works as a global safety layer
used for logging, crash reporting, and fallback handling.

Ссылка на видео этого руководства на сайте: borisproit.expert

Best practice:
use try/catch locally
and ExceptionHandler globally.

```
// Global fallback handler
val handler = CoroutineExceptionHandler { _, e ->
    println("Global handler: $e")   // logging / crashlytics
}

val scope = CoroutineScope(SupervisorJob() + Dispatchers.IO + handler)

// launch -> exceptions bubble to handler
scope.launch {
    try {
        val room = api.getRoomDetails("101")   // local handling
        println(room)
    } catch (e: IOException) {
        println("Handled locally: network error")
    }

    // This one NOT wrapped → goes to global handler
    error("Unexpected bug")
}
```

24. Как обрабатываются исключения в launch? How are exceptions handled in launch?

В launch исключения по умолчанию считаются окончанием корутины с ошибкой.

Если внутри launch нет try/catch, то:

- ошибка прерывает эту корутину
- ошибка отменяет родительский scope (если это обычный Job)
- если это корутина верхнего уровня, ошибка попадает в CoroutineExceptionHandler или в дефолтный обработчик (лог / crash)

Чтобы обработать ошибку «по-месту», в launch обычно используют try/catch внутри тела корутины.

Ссылка на видео этого руководства на сайте: borisproit.expert

---

In a launch coroutine, exceptions are treated as failures of the coroutine.

If there is no try/catch inside:

- the exception cancels that coroutine
- it propagates to the parent scope and may cancel it (for a normal Job)
- for a top-level coroutine, it is delivered to the CoroutineExceptionHandler or the default handler (log / crash)

To handle errors locally, you wrap the body of launch in try/catch.

```
val handler = CoroutineExceptionHandler { _, e ->
    println("Global handler: $e") // global fallback
}

val scope = CoroutineScope(SupervisorJob() + Dispatchers.IO + handler)

// launch: local handling with try/catch
scope.launch {
    try {
        val room = api.getRoomDetails("101") // suspend API call
        println("Loaded: ${room.name}")
    } catch (e: Exception) {
        println("Local error: ${e.message}") // handled here
    }
}

// launch without try/catch → goes to handler
scope.launch {
    error("Unexpected bug in launch") // will be caught by handler
}
```

25. Как обрабатываются исключения в async? How are exceptions handled in async?

Ссылка на видео этого руководства на сайте: borisproit.expert

В async исключения не выбрасываются сразу.
Они сохраняются внутри Deferred и выбрасываются в момент вызова await().

То есть:

- если ошибка произошла внутри async
- она «ждёт»
- и проявится только при await()

Если await() не вызвать — ошибка не всплывёт.

Чтобы обработать её, try/catch ставят вокруг await().

In async, exceptions are not thrown immediately.
They are stored inside the Deferred and are thrown when await() is called.

So:

- the error happens inside async
- but is only re-thrown at await()

If you never call await(), the exception will never surface.

You usually handle it with try/catch around await().

Ссылка на видео этого руководства на сайте: borisproit.expert

```
val scope = CoroutineScope(Dispatchers.IO)

// async throws only on await()
val deferred = scope.async {
    api.getRoomDetails("101")   // exception happens here
}

scope.launch {
    try {
        val room = deferred.await()   // exception re-thrown HERE
        println(room)
    } catch (e: Exception) {
        println("Handled async error: ${e.message}")
    }
}
```

26. Что происходит с sibling-корутинами при ошибке? What happens to sibling coroutines when one fails?

Это зависит от типа родительского scope.

1. Обычный Job / launch / coroutineScope

Если одна корутина упала —
все её “соседи” (siblings) тоже будут отменены.

То есть ошибка распространяется по семье.

2. SupervisorJob / supervisorScope

Если одна корутина упала —
соседние корутины продолжают работать.

Ошибка изолирована.

---

2) Short answer in English

It depends on the parent scope type.

Ссылка на видео этого руководства на сайте: borisproit.expert

1. Normal Job / launch / coroutineScope
If one child coroutine fails,
all sibling coroutines get cancelled too.

The failure propagates.

2. SupervisorJob / supervisorScope
If one child fails,
other siblings continue running.

The failure is isolated.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// ❌ Normal scope — siblings cancel each other
val normal = CoroutineScope(Job())

normal.launch {
    delay(200)
    println("Room pricing loaded")
}

normal.launch {
    error("Network crash") // cancels BOTH coroutines
}


// ✅ Supervisor scope — siblings are independent
val supervisor = CoroutineScope(SupervisorJob())

supervisor.launch {
    delay(200)
    println("Room availability loaded") // still runs
}

supervisor.launch {
    error("API failed") // only this one fails
}
```

E. Параллельность и async/await

27. Как запустить несколько задач параллельно? How do you run multiple tasks in parallel?

Ссылка на видео этого руководства на сайте: borisproit.expert

Для параллельного выполнения корутин используют async.
Каждая задача стартует одновременно, а результаты получают через await().
Так несколько API-запросов или операций могут выполняться параллельно.

To run tasks in parallel, you use async.
Each coroutine starts immediately, and results are retrieved using await().
This allows multiple API calls or operations to run concurrently.

```
val scope = CoroutineScope(Dispatchers.IO)

scope.launch {
    val roomDeferred = async { api.getRoomDetails("101") }
    val priceDeferred = async { api.getRoomPrice("101") }

    val room = roomDeferred.await()
    val price = priceDeferred.await()

    println("Room: ${room.name}, price: $price")
}
```

28. Что делает await() ?What does `await()` do?

await() приостанавливает корутину,
пока async-задача не завершится,
а затем возвращает её результат
или выбрасывает исключение, если задача упала.

Важно: поток при этом не блокируется.

---

await() suspends the coroutine
until the async task completes,
then returns its result
or throws the exception from that task.

Important: the thread is not blocked.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
val scope = CoroutineScope(Dispatchers.IO)

scope.launch {
    val roomDeferred = async { api.getRoomDetails("101") }

    val room = roomDeferred.await()   // suspend here

    println("Loaded room: ${room.name}")
}
```

29. **Что будет, если забыть вызвать await()? What happens if you forget to call `await()`?**

Если забыть вызвать await():

- async-корутина всё равно выполнится
- но её результат будет потерян
- а ошибка внутри неё не всплывёт
(останется внутри Deferred)

То есть программа «работает»,
но вы не получаете данные и не узнаёте об ошибках.

If you forget to call await():

- the async coroutine will still run
- but its result will be discarded
- and any exception inside it will not surface

So the program continues running,
but you lose the result and error reporting.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
val deferred = scope.async {
    api.getRoomDetails("101")  // runs anyway
}

// ❌ forgot await()
// nothing crashes
// but result is never used
```

30. Чем параллельность отличается от последовательного выполнения? What is the difference between parallel and sequential execution?

Последовательное выполнение —
задачи выполняются по очереди:
вторая начинается только после завершения первой.

Параллельное выполнение —
несколько задач запускаются одновременно
(например, через async)
и выполняются независимо, пока вы ждёте их результат через await().

Обычно параллельность экономит время
при сетевых и I/O операциях.

Sequential execution means
tasks run one after another —
the next task starts only when the previous one finishes.

Parallel execution means
multiple tasks run at the same time
(e.g., using async)
and you wait for them later using await().

Parallelism usually saves time
for network and I/O operations.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
val scope = CoroutineScope(Dispatchers.IO)

// ❌ Sequential
scope.launch {
    val room = api.getRoomDetails("101")
    val price = api.getRoomPrice("101")
}

// ✅ Parallel
scope.launch {
    val roomDeferred = async { api.getRoomDetails("101") }
    val priceDeferred = async { api.getRoomPrice("101") }

    val room = roomDeferred.await()
    val price = priceDeferred.await()
}
```

31. Чем параллельность отличается от многопоточности? What is the difference between parallelism and multithreading?

Параллельность в корутинах —
это когда несколько задач *логически* выполняются одновременно,
но они могут работать даже внутри одного потока,
потому что корутины просто приостанавливаются и возобновляются.

Многопоточность —
это когда реально используются разные системные потоки.

Главная разница:

Параллельность = способ организации работы
Многопоточность = механизм на уровне ОС

Корутины могут давать параллельность
без дополнительных потоков.

Ссылка на видео этого руководства на сайте: borisproit.expert

Parallelism in coroutines means
multiple tasks appear to run at the same time,
even if they share the same thread,
because coroutines suspend and resume.

Multithreading means
multiple OS threads are used in reality.

Key idea:

Parallelism = logical concurrency
Multithreading = physical threads

Coroutines may achieve parallelism
without creating new threads.

```
// parallel – same thread, suspending
scope.launch {
    val a = async { api.loadRooms() }   // may suspend
    val b = async { api.loadGuests() }  // may suspend
    a.await()
    b.await()
}

// multithreading – different threads
withContext(Dispatchers.IO) {
    // runs on worker thread
}
```

F. Android lifecycle и корутины

32. Чем viewModelScope отличается от lifecycleScope? What is the difference between viewModelScope and lifecycleScope?

viewModelScope живёт вместе с ViewModel
и продолжает работу при повороте экрана.
Корутины отменяются только в onCleared().

Ссылка на видео этого руководства на сайте: borisproit.expert

lifecycleScope привязан к Activity/Fragment
и отменяется, когда UI уничтожается
(например, при закрытии экрана).

Идея:
viewModelScope — для данных и бизнес-логики
lifecycleScope — для UI

---

viewModelScope lives as long as the ViewModel,
so it survives configuration changes.
Coroutines are cancelled only in onCleared().

lifecycleScope is tied to Activity/Fragment lifecycle
and is cancelled when the UI is destroyed.

Idea:
viewModelScope — data/business logic
lifecycleScope — UI work

```
// ViewModel – survives rotation
class HotelViewModel : ViewModel() {
    fun loadRooms() = viewModelScope.launch {
        roomsState.value = api.getRooms()
    }
}

// Activity – UI lifecycle
lifecycleScope.launch {
    viewModel.roomsState.collect { showRooms(it) }
}
```

33. **Почему нельзя выполнять долгие операции на Dispatchers.Main? Why should long-running operations not run on `Dispatchers.Main`?**

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// ❌ Wrong — freezes UI
lifecycleScope.launch(Dispatchers.Main) {
    val rooms = api.getRooms() // long work
    showRooms(rooms)
}

// ✅ Correct
lifecycleScope.launch(Dispatchers.Main) {
    val rooms = withContext(Dispatchers.IO) {
        api.getRooms()
    }
    showRooms(rooms)
}
```

Dispatchers.Main — это UI-поток.
Если выполнять на нём долгие операции (сеть, БД, вычисления) — он блокируется, и:

- интерфейс зависает
- анимации останавливаются
- нажатия не обрабатываются
- система может показать ANR (App Not Responding)

Поэтому долгие операции нужно переносить в Dispatchers.IO или Dispatchers.Default.

Dispatchers.Main is the UI thread.
If you run long-running work on it (network, DB, CPU tasks), the UI freezes, animations stop, user input is blocked, and the app may get an ANR.

So long-running work should run on Dispatchers.IO or Dispatchers.Default.

Ссылка на видео этого руководства на сайте: borisproit.expert

34. Почему GlobalScope — плохая идея?Why is `GlobalScope` a bad idea?

```
// ❌ Bad: keeps running even after screen is closed
GlobalScope.launch {
    api.syncHotelPrices()
}

// ✅ Correct: tied to ViewModel lifecycle
class HotelViewModel : ViewModel() {
    fun sync() = viewModelScope.launch {
        api.syncHotelPrices()
    }
}
```

GlobalScope живёт столько же, сколько всё приложение
и не привязан к lifecycle.
Из-за этого:

- корутины могут утекать в память
- они продолжают работать после закрытия экрана
- отмена не контролируется
- ошибки могут крашить приложение
- сложно дебажить

Лучше использовать
viewModelScope, lifecycleScope или свой CoroutineScope с Job.

GlobalScope lives for the entire lifetime of the app
and is not lifecycle-aware.
This leads to:

- memory leaks
- coroutines running after UI is gone
- no proper cancellation
- app-crashing exceptions
- debugging complexity

Prefer viewModelScope, lifecycleScope,
or a custom scope with a Job.

35. Где правильнее запускать корутины — во ViewModel или Activity? Where should you launch coroutines — in a ViewModel or in an Activity?

```
// ViewModel – runs business logic
class HotelViewModel : ViewModel() {
    val roomsState = MutableStateFlow<List<Room>>(emptyList())

    fun loadRooms() = viewModelScope.launch {
        roomsState.value = api.getRooms()
    }
}

// Activity – only UI
lifecycleScope.launch {
    viewModel.roomsState.collect { rooms ->
        showRooms(rooms)
    }
}
```

Обычно корутины лучше запускать во ViewModel,
потому что ViewModel переживает повороты экрана
и не привязана жёстко к UI.

ViewModel — для
✔ загрузки данных

Ссылка на видео этого руководства на сайте: borisproit.expert

✔ вызовов API
✔ работы с БД
✔ бизнес-логики

Activity / Fragment — для
✔ отображения UI
✔ подписок на StateFlow / LiveData
✔ реакций на события

In most cases, coroutines should run in the ViewModel,
because ViewModel survives configuration changes
and is not tightly coupled to UI.

Use ViewModel for:
✔ data loading
✔ API calls
✔ DB work
✔ business logic

Use Activity / Fragment for:
✔ UI updates
✔ collecting Flow / LiveData
✔ handling user interactions

36. Когда нужно использовать Dispatchers.Main? When should `Dispatchers.Main` be used?

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// UI thread
lifecycleScope.launch(Dispatchers.Main) {

    // background work
    val rooms = withContext(Dispatchers.IO) {
        api.getRooms()
    }

    // back to UI
    showRooms(rooms)
}
```

Dispatchers.Main используют тогда,
когда работа должна выполняться в UI-потоке:

- обновление интерфейса
- работа с View
- сбор Flow для UI
- навигация
- показ диалогов и т.д.

Долгие операции на Main делать нельзя —
их нужно выносить в IO / Default.

Use Dispatchers.Main when the work must run
on the UI thread:

- updating UI
- interacting with Views
- collecting Flow for UI

Ссылка на видео этого руководства на сайте: borisproit.expert

- navigation
- showing dialogs, etc.

Do not run long-running work on Main —
use IO / Default instead.

🟡 ВАЖНО ДЛЯ MIDDLE (часто спрашивают)

G. Flow — основы

37. Что такое Flow? What is Flow?

```
// Repository — Flow runs only when collected
fun loadRooms(): Flow<List<Room>> = flow {
    emit(api.getRooms())   // API request happens here
}

// UI — collection triggers work
lifecycleScope.launch {
    loadRooms().collect { rooms ->
        showRooms(rooms)
    }
}
```

Flow — это холодный асинхронный поток данных,
который начинает свою работу только тогда,
когда на него подписываются через collect().

Flow не хранит данные —
он просто выполняет логику (например, API-запрос, чтение из БД)
и передаёт значения подписчику по мере готовности.

Ссылка на видео этого руководства на сайте: borisproit.expert

Если подписаться несколько раз —
логика выполнится каждый раз заново.

Flow is a cold asynchronous data stream
that starts its work only when collected using collect().

It does not store data —
instead, it runs some logic (like API calls or DB queries)
and emits values to the collector as they become available.

If collected multiple times,
the work runs again each time.

38. **Чем Flow отличается от suspend-функции? What is the difference between Flow and a suspend function?**

```
// suspend – one result
suspend fun loadRoom(): Room {
    return api.getRoomDetails("101")
}

// Flow – multiple values over time
fun observeRooms(): Flow<List<Room>> = flow {
    emit(api.getRooms())
    emit(api.refreshRooms())
}
```

suspend-функция возвращает один результат и завершается.
Flow может возвращать много значений со временем и остаётся активным, пока его собирают.

Ссылка на видео этого руководства на сайте: borisproit.expert

Главное отличие:
suspend = один результат
Flow = поток результатов

A suspend function returns a single result and then finishes.
A Flow can emit multiple values over time and stays active while it is collected.

Key difference:
suspend = single result
Flow = stream of results

39. Что делает оператор collect? What does the `collect` operator do?

```
lifecycleScope.launch {
    roomsFlow.collect { rooms ->
        showRooms(rooms)
    }
}
```

collect подписывается на Flow и запускает его выполнение.
Пока collect активен, он получает все значения, которые Flow эмитит.
Без collect Flow не выполняется вообще.

---

collect subscribes to a Flow and starts its execution.
While collect is active, it receives all values emitted by the Flow.
Without collect, a Flow does nothing.

Ссылка на видео этого руководства на сайте: borisproit.expert

40. **Чем Flow отличается от StateFlow? What is the difference between Flow and StateFlow?**

```
// Flow – runs on each collect
fun loadRooms(): Flow<List<Room>> = flow {
    emit(api.getRooms())
}

// StateFlow – holds latest state
class RoomsViewModel : ViewModel() {
    private val _rooms = MutableStateFlow<List<Room>>(emptyList())
    val rooms = _rooms

    fun refresh() = viewModelScope.launch {
        _rooms.value = api.getRooms()
    }
}
```

Flow — это холодный поток данных, который не хранит значения и начинает работу только при collect(). Каждый новый сбор может заново запустить логику.

StateFlow — это горячий поток состояния, который всегда хранит последнее значение и сразу отдаёт его новому подписчику.

Ключевая идея:
Flow — «выполнить и отдать»
StateFlow — «хранить и обновлять»

Flow is a cold data stream that does not store values and starts only when collected. Each new collection may re-run the logic.

StateFlow is a hot state holder that keeps the latest value and immediately emits it to new collectors.

Key idea:
Flow = compute & emit
StateFlow = hold & update

Ссылка на видео этого руководства на сайте: borisproit.expert

H. Flow в UI

42. Что такое StateFlow? What is `StateFlow`?

```
class RoomsViewModel : ViewModel() {
    private val _state = MutableStateFlow<List<Room>>(emptyList())
    val state = _state

    fun load() = viewModelScope.launch {
        _state.value = api.getRooms()
    }
}
```

StateFlow — это горячий поток состояния, который всегда хранит последнее значение и немедленно отдаёт его каждому новому подписчику.
Он используется для представления текущего состояния (чаще всего UI-state) и безопасно работает с корутинами и отменой.

StateFlow is a hot state holder that always keeps the latest value and immediately emits it to any new collector.
It is commonly used to represent current state (especially UI state) and works safely with coroutines and cancellation.

Ссылка на видео этого руководства на сайте: borisproit.expert

43. Что такое SharedFlow? What is SharedFlow?

```
class RoomsViewModel : ViewModel() {
    private val _events = MutableSharedFlow<String>()
    val events = _events

    fun bookRoom() = viewModelScope.launch {
        _events.emit("Room booked")
    }
}

// UI
lifecycleScope.launch {
    viewModel.events.collect { message ->
        showToast(message)
    }
}
```

SharedFlow — это горячий поток событий, предназначенный для рассылки событий нескольким подписчикам.
Он не хранит состояние, а передаёт события тем, кто подписан в момент эмита.
Обычно используется для одноразовых событий: навигация, тосты, сообщения об ошибках.

SharedFlow is a hot event stream used to broadcast events to multiple collectors.
It does not represent state and delivers events only to collectors that are active at emission time.
It is commonly used for one-time events like navigation, toasts, or error messages.

44. Чем SharedFlow отличается от StateFlow? What is the difference between SharedFlow and StateFlow?

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// StateFlow – UI state
class RoomsViewModel : ViewModel() {
    val rooms = MutableStateFlow<List<Room>>(emptyList())
}

// SharedFlow – UI events
class EventsViewModel : ViewModel() {
    val events = MutableSharedFlow<String>()
}
```

StateFlow — это поток состояния, который всегда хранит последнее значение и немедленно отдаёт его новому подписчику. Он используется для представления текущего состояния UI.

SharedFlow — это поток событий, который по умолчанию не хранит значения и передаёт события только активным подписчикам. Он используется для одноразовых действий: навигация, тосты, ошибки.

Ключевая идея:
StateFlow — состояние
SharedFlow — события

StateFlow is a state holder that always keeps the latest value and immediately emits it to new collectors. It represents current UI state.

SharedFlow is an event stream that does not store values by default and delivers events only to active collectors. It is used for one-time events like navigation or messages.

Key idea:
StateFlow — state
SharedFlow — events

I. Конкурентный доступ и безопасность

Ссылка на видео этого руководства на сайте: borisproit.expert

47. Какие проблемы возникают при работе с shared mutable state? What problems arise when working with shared mutable state?

```
var availableRooms = 10

viewModelScope.launch {
    availableRooms -= 1
}

viewModelScope.launch {
    availableRooms -= 1
}
```

Shared mutable state — это общее изменяемое состояние, к которому одновременно обращаются несколько корутин.
Основные проблемы: race conditions, неконсистентные данные, потеря обновлений и трудноуловимые баги, зависящие от порядка выполнения.

Без синхронизации результат работы программы становится непредсказуемым.

Shared mutable state means shared mutable data accessed by multiple coroutines at the same time.
The main problems are race conditions, inconsistent data, lost updates, and hard-to-debug timing issues.

Without proper synchronization, program behavior becomes unpredictable.

Ссылка на видео этого руководства на сайте: borisproit.expert

48. Что такое Mutex? What is a Mutex?

```
val mutex = Mutex()
var availableRooms = 10

viewModelScope.launch {
    mutex.withLock {
        availableRooms -= 1
    }
}
```

Mutex — это механизм синхронизации для корутин, который гарантирует, что только одна корутина в один момент времени может выполнять защищённый участок кода.
Он используется для безопасной работы с shared mutable state.

Mutex is a coroutine synchronization primitive that ensures only one coroutine at a time can execute a critical section of code.
It is used to safely protect shared mutable state.

49. Почему иммутабельность предпочтительнее блокировок? Why is immutability preferred over locks?

```
data class Room(val id: String, val isAvailable: Boolean)

viewModelScope.launch {
    val room = Room(id = "101", isAvailable = true)
    // safe to share across coroutines
}
```

Иммутабельность предпочтительнее, потому что неизменяемые данные нельзя повредить при конкурентном доступе.
Если объект нельзя изменить, его можно безопасно читать из нескольких корутин без Mutex, блокировок и race conditions.

Ссылка на видео этого руководства на сайте: borisproit.expert

---

Immutability is preferred because immutable data cannot be corrupted by concurrent access.
If an object cannot be modified, it can be safely shared between coroutines without locks or synchronization.

50. Что такое thread confinement? What is *thread confinement*?

```
var availableRooms = 10

viewModelScope.launch(Dispatchers.Main) {
    availableRooms -= 1 // always on Main thread
}
```

Здесь состояние изменяется только на Dispatchers.Main, поэтому доступ безопасен без Mutex.

Thread confinement — это подход, при котором изменяемое состояние доступно только из одного потока.
Если все операции с данными выполняются на одном и том же потоке, синхронизация и блокировки не нужны.

Thread confinement is a concurrency approach where mutable state is accessed from a single thread only.
When all operations happen on the same thread, no locks or synchronization are required.

Ссылка на видео этого руководства на сайте: borisproit.expert

J. Cancellation & Timeout

51. Что такое withTimeout? What is `withTimeout`?

```
viewModelScope.launch {
    withTimeout(2_000) {
        api.loadRooms() // cancelled if takes more than 2 seconds
    }
}
```

Здесь корутина будет автоматически отменена, если loadRooms() выполняется слишком долго.

withTimeout — это suspend-функция, которая ограничивает время выполнения корутины.
Если код внутри не успевает выполниться за заданное время, корутина отменяется с исключением TimeoutCancellationException.

---

withTimeout is a suspend function that limits how long a coroutine is allowed to run.
If the block does not finish in time, the coroutine is cancelled with a TimeoutCancellationException.

52. Что такое withTimeoutOrNull? What is `withTimeoutOrNull`?

```
viewModelScope.launch {
    val rooms = withTimeoutOrNull(2_000) {
        api.loadRooms()
    }

    if (rooms == null) {
        showTimeoutMessage()
    }
}
```

Здесь при превышении времени приложение не падает — вместо этого приходит null.

Ссылка на видео этого руководства на сайте: borisproit.expert

withTimeoutOrNull — это suspend-функция, которая ограничивает время выполнения корутины, но не выбрасывает исключение.
Если время истекло, она возвращает null и корутина продолжается дальше.

---

withTimeoutOrNull is a suspend function that limits coroutine execution time without throwing an exception.
If the timeout is reached, it returns null and the coroutine continues.

53. Что такое CancellationException? What is `CancellationException`?

```
viewModelScope.launch {
    try {
        val data = repository.loadData()
        show(data)
    } catch (e: CancellationException) {
        throw e // keep normal coroutine cancellation
    } catch (e: Exception) {
        showError()
    }
}
```

Здесь отмена — это нормальное поведение, а не ошибка.

CancellationException — это служебный сигнал корутин, а не ошибка приложения.
Она означает: *«эту корутину нужно корректно остановить»* и возникает только при отмене (cancel, lifecycle, ViewModel).
Её нельзя глотать, при перехвате её нужно пробрасывать дальше.

CancellationException is a control signal, not an application error.
It means *"this coroutine should stop execution"* and appears only on cancellation.
It must not be swallowed and should be re-thrown if caught.

54. Как корректно освобождать ресурсы при отмене? How do you release resources correctly on cancellation?

Ссылка на видео этого руководства на сайте: borisproit.expert

```
viewModelScope.launch {
    val socket = openSocket()
    try {
        api.streamUpdates(socket)
    } finally {
        withContext(NonCancellable) {
            socket.close()
        }
    }
}
```

Что это означает буквально

“Даже если корутина отменена —
разреши этому блоку выполниться до конца”

Что меняется

- отмена временно игнорируется
- код внутри гарантированно выполняется
- после выхода — отмена снова действует

finally:

- гарантирует вход в блок
- ❌ не гарантирует, что suspend-код внутри выполнится

NonCancellable:

- гарантирует выполнение suspend-кода

👉 Именно поэтому они используются вместе

Use `try / finally`: `finally` runs on normal completion and on cancellation.

Ссылка на видео этого руководства на сайте: borisproit.expert

If cleanup needs suspend calls, wrap them in `withContext(NonCancellable)` so cancellation won't interrupt the cleanup.

55. **Что выполняется в finally внутри корутины? What runs inside `finally` in a coroutine?**

```
viewModelScope.launch {
    try {
        api.loadRooms()
    } finally {
        withContext(NonCancellable) {
            closeConnection()
        }
    }
}
```

Здесь finally выполнится в любом случае, а NonCancellable позволяет корректно завершить очистку.

Блок finally выполняется всегда — и при успешном завершении корутины, и при исключении, и при отмене.
Он используется для освобождения ресурсов и cleanup.
Важно: при отмене корутина уже отменена, поэтому suspend-вызовы в finally запрещены, если не использовать NonCancellable.

The finally block always runs — on normal completion, on exception, and on cancellation.
It is used for cleanup and resource release.
Important: on cancellation the coroutine is already cancelled, so suspending calls are not allowed in finally unless wrapped with NonCancellable.

K. Supervisor и ошибка-изоляция

56. **Что такое SupervisorJob? What is a `SupervisorJob`?**

Ссылка на видео этого руководства на сайте: borisproit.expert

```
val scope = CoroutineScope(SupervisorJob() + Dispatchers.IO)

scope.launch {
    api.loadRooms() // fails
}

scope.launch {
    api.loadGuests() // continues
}
```

Здесь ошибка в одной корутине не остановит выполнение другой.

SupervisorJob — это специальный Job, при котором ошибка дочерней корутины не отменяет родительскую и другие дочерние корутины.
Он используется, когда задачи независимы друг от друга и падение одной не должно ломать остальные.

SupervisorJob is a special Job where a failure in a child coroutine does not cancel its parent or sibling coroutines.
It is used when child tasks are independent and one failure should not affect others.

57. **Чем supervisorScope отличается от coroutineScope? What is the difference between `supervisorScope` and `coroutineScope`?**

```
suspend fun loadHotelData() = supervisorScope {
    launch { api.loadRooms() }    // may fail
    launch { api.loadGuests() }   // still runs
}
```

coroutineScope — при ошибке одной дочерней корутины отменяются все остальные и сам scope.
supervisorScope — при ошибке отменяется только упавшая корутина, остальные продолжают работать.

Ссылка на видео этого руководства на сайте: borisproit.expert

coroutineScope — if one child coroutine fails, all other children and the scope are cancelled.
supervisorScope — if one child fails, only that child is cancelled, others continue running.

58. **Как работает propagation исключений с Supervisor? How does exception propagation work with `Supervisor`?**

```
val scope = CoroutineScope(SupervisorJob() + Dispatchers.IO)

scope.launch {
    error("Rooms API failed") // cancelled only here
}

scope.launch {
    api.loadGuests() // still runs
}
```

Здесь исключение не “поднимается” вверх и не отменяет соседние корутины.

С SupervisorJob / supervisorScope исключения НЕ распространяются вверх и к sibling-корутинам.
Ошибка отменяет только ту корутину, где она произошла.
Родитель и другие дочерние корутины продолжают работать.

With SupervisorJob / supervisorScope, exceptions do NOT propagate to the parent or sibling coroutines.
The failure cancels only the coroutine where it occurred.
The parent and other children keep running.

59. **Почему supervisor полезен для UI-логики? Why is `supervisor` useful for UI logic?**

Supervisor полезен для UI, потому что ошибка в одной UI-задаче не ломает весь экран.
Загрузка данных, анимации, таймеры и события UI часто независимы, и падение одной операции не должно отменять остальные.

Ссылка на видео этого руководства на сайте: borisproit.expert

Supervisor is useful for UI logic because a failure in one UI task does not cancel the whole screen.
UI operations are often independent, and one failure should not stop others.

🔵 ДЛЯ УВЕРЕННОГО MIDDLE / SENIOR

L. Advanced Flow

61. Что такое backpressure? What is backpressure?

Backpressure — это ситуация, когда производитель данных генерирует значения быстрее, чем потребитель успевает их обрабатывать.
Без контроля backpressure это приводит к переполнению памяти, лагам или потере данных.

Backpressure is a situation where the data producer emits values faster than the consumer can process them.
Without handling backpressure, this can cause memory issues, UI lag, or dropped data.

62. Для чего используется оператор buffer()? What is the `buffer()` operator used for?

```
roomsFlow
    .buffer()
    .collect { rooms ->
        renderRooms(rooms)
    }
```

Здесь buffer() позволяет roomsFlow продолжать эмитить данные, даже если renderRooms() обрабатывает их медленно.

buffer() используется для управления backpressure — он позволяет Flow временно накапливать элементы, если производитель работает быстрее, чем потребитель.
Это помогает избежать блокировок, лагов и потери данных.

Ссылка на видео этого руководства на сайте: borisproit.expert

buffer() is used to handle backpressure by allowing a Flow to temporarily store emitted values when the producer is faster than the consumer.
It helps prevent blocking, UI lag, and dropped emissions.

63. **Чем conflate() отличается от buffer()? What is the difference between `conflate()` and `buffer()`?**

```
roomsFlow
    .conflate()
    .collect { rooms ->
        renderRooms(rooms)
    }
```

Здесь, если данные обновляются часто, UI будет получать только последнее состояние, пропуская устаревшие обновления.

buffer() сохраняет все элементы во временном буфере, если потребитель не успевает.
conflate() отбрасывает промежуточные элементы и оставляет только последний, если потребитель медленный.

Использование:

- buffer() — когда важны все значения
- conflate() — когда важен только актуальный результат

---

buffer() keeps all emitted values in a buffer when the consumer is slow.
conflate() drops intermediate values and keeps only the latest one.

Use cases:

64. buffer() — when every value matters

65. conflate() — when only the latest value matters

66. **Чем collectLatest отличается от collect? What is the difference between `collectLatest` and `collect`?**

Ссылка на видео этого руководства на сайте: borisproit.expert

```
roomsFlow.collectLatest { rooms ->
    renderRooms(rooms) // previous render is cancelled if new data arrives
}
```

Здесь при частых обновлениях UI не тратит время на устаревшие данные — обрабатывается только актуальное состояние.

collect обрабатывает все элементы по очереди — каждый элемент должен быть полностью обработан, прежде чем начнётся следующий.
collectLatest отменяет обработку предыдущего элемента, если приходит новый, и начинает работать только с последним значением.

Использование:

- collect — когда важны все значения
- collectLatest — когда важен только последний результат

---

collect processes every emitted value sequentially — each one is fully handled before the next starts.
collectLatest cancels the previous processing when a new value arrives and handles only the latest one.

Use cases:

- collect — when every value matters
- collectLatest — when only the latest value matters

67. Чем отличаются холодные и горячие потоки? What is the difference between cold and hot flows?

```
// Cold Flow
val roomsFlow = flow {
    emit(api.loadRooms())
}

// Hot Flow
val roomsState = MutableStateFlow(emptyList<Room>())
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Flow — холодный: запускается при collect.

StateFlow / SharedFlow — горячие: живут независимо от подписчиков.

Холодные потоки (cold) начинают работу только при подписке — каждый подписчик запускает поток заново и получает свою последовательность данных.
Горячие потоки (hot) работают независимо от подписчиков — данные эмитятся постоянно, а подписчики получают только то, что происходит после подписки (или последнее состояние, если оно хранится).

Cold streams start emitting only when collected — each collector gets its own execution and data sequence.
Hot streams emit data independently of collectors — collectors receive values from a shared, ongoing stream.

M. Практическая архитектура

66. Почему корутины лучше callback-подхода? Why are coroutines better than callbacks?

```
viewModelScope.launch {
    try {
        val room = api.getRoomDetails("101")
        show(room)
    } catch (e: Exception) {
        showError()
    }
}
```

Корутины позволяют писать асинхронный код как обычный последовательный, без вложенных колбэков. Это даёт:

- более читаемый код (без “callback hell”)
- обычный try/catch для ошибок

Ссылка на видео этого руководства на сайте: borisproit.expert

- структурированную отмену через Job и CoroutineScope
- понятное управление жизненным циклом (особенно в Android)

Coroutines let you write async code in a sequential, readable style, avoiding nested callbacks. They provide:

- cleaner code (no callback hell)
- normal try/catch error handling
- structured cancellation via Job and CoroutineScope
- lifecycle-friendly async code in Android

67. Чем корутины отличаются от RxJava? How are coroutines different from RxJava?

Корутины — это встроенный в Kotlin механизм асинхронности, ориентированный на простой, последовательный код и structured concurrency.
RxJava — это реактивная библиотека, ориентированная на потоки данных и операторы, с более сложной моделью и большим порогом входа.

Корутины проще для большинства задач Android (сеть, БД, UI),
RxJava мощнее для сложных реактивных сценариев.

Coroutines are a Kotlin language-level async mechanism focused on sequential code and structured concurrency.
RxJava is a reactive library focused on data streams and operators, with higher complexity and a steeper learning curve.

Coroutines are simpler for most Android use cases,
RxJava is more powerful for complex reactive pipelines.

68. Когда лучше использовать Flow, а когда suspend-функции? When should you use `Flow` vs `suspend` functions?

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// suspend — one result
suspend fun loadRoom(roomId: String): Room =
    api.getRoomDetails(roomId)

// Flow — multiple updates
fun observeRooms(): Flow<List<Room>> =
    roomDao.observeRooms()
```

Используй suspend-функции, когда нужен один результат: “сделать запрос и получить ответ”.
Используй Flow, когда нужны много значений во времени: “наблюдать обновления/стрим событий”.

Правило:
suspend = one-shot
Flow = stream / updates

Use suspend functions when you need a single result (“request → response”).
Use Flow when you need multiple values over time (“observe updates / events”).

Rule:
suspend = one-shot
Flow = stream / updates

69. Почему structured concurrency предотвращает утечки? Why does structured concurrency prevent leaks?

Structured concurrency предотвращает утечки, потому что каждая корутина привязана к своему CoroutineScope и его жизненному циклу.
Когда scope отменяется (Activity уничтожена, ViewModel очищен), все дочерние корутины автоматически отменяются, и “висящие” фоновые задачи не остаются.

Structured concurrency prevents leaks because every coroutine is bound to a CoroutineScope and its lifecycle.
When the scope is cancelled (Activity destroyed, ViewModel cleared), all child coroutines are automatically cancelled, so no background work keeps running.

Ссылка на видео этого руководства на сайте: borisproit.expert

70. Как тестировать корутины? How do you test coroutines?

```
@Test
fun loadsRooms() = runTest {
    val repo = FakeRoomsRepo()
    val vm = RoomsViewModel(repo)

    vm.load()
    advanceUntilIdle()

    assertEquals(3, vm.state.value.size)
}
```

Корутины тестируют через kotlinx-coroutines-test:

- запускают тест в runTest
- runTest — это специальный блок, внутри которого:
- ты можешь вызывать suspend функции напрямую
- создаётся тестовый scope для корутин

Зачем это нужно

- @Test функция не может быть suspend
- но внутри runTest ты ведёшь себя как будто в корутине

Ссылка на видео этого руководства на сайте: borisproit.expert

-

- используют тестовый dispatcher/виртуальное время

Что такое test dispatcher

Это диспетчер, который:

- выполняет корутины под контролем теста
- работает вместе с тестовым scheduler
- позволяет “перематывать” время

Что такое виртуальное время

Это когда delay(10_000) не ждёт 10 секунд,
а “ставится в очередь” до тех пор, пока ты не скажешь тесту:

- перемотай время
- или выполни всё, что ожидает
-
-

- контролируют delay через advanceTimeBy / advanceUntilIdle
  - advanceTimeBy(ms)
  - Перематывает виртуальное время вперёд на указанное количество миллисекунд.
  - Пример: в коде есть delay(1000)
  - delay(1000) не ждёт реально.

Ссылка на видео этого руководства на сайте: borisproit.expert

- ```
  @Test
  fun testDelay() = runTest {
      var done = false

      launch {
          delay(1000)
          done = true
      }

      assertFalse(done)

      advanceTimeBy(1000) // перемотали 1 секунду

      assertTrue(done)
  }
  ```

- проверяют результат без реального ожидания и без реальных потоков

Без реального ожидания

Потому что:

- вместо реального `delay()` ты управляешь виртуальным временем

✅ тест выполняется за миллисекунды
❌ не надо `Thread.sleep()`

Без реальных потоков

Потому что:

- `test dispatcher` не требует настоящего `Dispatchers.IO/Main`
- всё выполняется “управляемо” внутри тестового окружения

Это делает тест:
  ```

Ссылка на видео этого руководства на сайте: borisproit.expert

- быстрым
- детерминированным
- стабильным
-

Test coroutines with kotlinx-coroutines-test:

- run tests with runTest
- use test dispatchers / virtual time
- control delay via advanceTimeBy / advanceUntilIdle
- assert results without real waiting or real threads

🔴 ПРОДВИНУТЫЙ УРОВЕНЬ (для глубокой подготовки)

N. Dispatcher & threading

71. Как работает Dispatchers.Default? How does `Dispatchers.Default` work?

```
viewModelScope.launch(Dispatchers.Default) {
    val prices = calculateRoomPrices() // CPU-heavy work
    withContext(Dispatchers.Main) {
        showPrices(prices)
    }
}
```

Dispatchers.Default предназначен для CPU-нагруженных задач.
Он использует общий пул потоков, обычно размером по числу ядер процессора, и распределяет корутины так, чтобы эффективно загружать CPU без блокировок.

Ссылка на видео этого руководства на сайте: borisproit.expert

Типичные задачи для Dispatchers.Default:

- сложные вычисления и алгоритмы
- обработка больших коллекций (map/filter/reduce)
- парсинг и преобразование данных (JSON → модели, агрегации)
- сортировки, поиск, хэширование
- компрессия/декомпрессия данных
- вычисление цен, статистики, рейтингов

Dispatchers.Default is designed for CPU-intensive work.
It uses a shared thread pool, typically sized to the number of CPU cores, and schedules coroutines to efficiently utilize the CPU.

Typical use cases for Dispatchers.Default:

- heavy computations and algorithms
- large collection processing (map/filter/reduce)
- data parsing and transformations
- sorting, searching, hashing
- compression/decompression
- price calculations, statistics, scoring

72. Когда стоит использовать Dispatchers.IO? When should `Dispatchers.IO` be used?

Ссылка на видео этого руководства на сайте: borisproit.expert

```
viewModelScope.launch(Dispatchers.IO) {
    val rooms = api.loadRooms() // I/O work
    withContext(Dispatchers.Main) {
        showRooms(rooms)
    }
}
```

Dispatchers.IO предназначен для блокирующих операций ввода-вывода.
Он использует расширяемый пул потоков, чтобы такие операции не блокировали CPU и UI.

Типичные задачи для Dispatchers.IO:

- сетевые запросы (HTTP, REST, GraphQL)
- работа с базой данных
- чтение и запись файлов
- обращения к диску
- блокирующие SDK и legacy API
- сериализация / десериализация данных

Dispatchers.IO is designed for blocking I/O operations.
It uses a shared, expandable thread pool so blocking work does not freeze CPU or UI threads.

Typical use cases for Dispatchers.IO:

- network requests (HTTP, REST, GraphQL)
- database operations
- file read/write
- disk access
- blocking SDKs and legacy APIs
- serialization / deserialization

Ссылка на видео этого руководства на сайте: borisproit.expert

73. Что делает Dispatchers.Unconfined? What does `Dispatchers.Unconfined` do?

```
viewModelScope.launch(Dispatchers.Unconfined) {
    logThread("start")
    delay(100)
    logThread("after delay")
}
```

Dispatchers.Unconfined запускает корутину в текущем потоке до первой точки приостановки.
После suspend корутина возобновляется в том потоке, который решит suspend-функция, поэтому поток выполнения не гарантирован.
Используется редко — в основном для тестов или очень специфичных случаев.

Dispatchers.Unconfined starts a coroutine in the current thread until the first suspension point.
After suspension, it resumes on whatever thread the suspending function chooses, so thread confinement is not guaranteed.
It is rarely used, mostly for tests or very special cases.

74. Когда нужен кастомный dispatcher? When do you need a custom dispatcher?

Ссылка на видео этого руководства на сайте: borisproit.expert

```
class PaymentsViewModel : ViewModel() {

    private val paymentsDispatcher =
        Executors.newSingleThreadExecutor().asCoroutineDispatcher()

    override fun onCleared() {
        paymentsDispatcher.close() // ✅ остановит executor
    }

    fun startPayments() {
        viewModelScope.launch(paymentsDispatcher) {
            processPayments()
        }
    }
}
```

1) Что делает Executors.newSingleThreadExecutor()

Это Java ExecutorService, который создаёт пул из ровно одного потока.

- Внутри есть очередь задач
- Все задачи выполняются строго по одной
- В один момент времени работает только один Runnable

То есть это буквально: “один рабочий поток + очередь”.

---

2) Что делает .asCoroutineDispatcher()

asCoroutineDispatcher() — это адаптер:
он превращает Java Executor в CoroutineDispatcher.

То есть теперь корутины могут сказать:

“Запусти мой код через этот executor”.

Получается кастомный dispatcher, который:

Ссылка на видео этого руководства на сайте: borisproit.expert

- отправляет выполнение корутин в тот самый один поток
- гарантирует последовательность (по очереди)
- изолирует выполнение от остальных Dispatchers.Default/IO/Main

---

3) Что делает viewModelScope.launch(singleThreadDispatcher)

viewModelScope.launch(...):

- создаёт новую корутину, привязанную к ViewModel
- при onCleared() scope отменится → корутина отменится

А параметр singleThreadDispatcher означает:

весь код внутри корутины будет исполняться не на Main, и не на IO,
а на нашем одном выделенном потоке.

---

4) Что значит “isolated, one thread” на практике

Реальная ситуация, где это нужно

Например: processPayments():

- читает/пишет общие структуры данных (кеш, очередь платежей)
- обновляет локальную БД
- общается с SDK оплаты
- имеет требование “обрабатывать платежи строго последовательно”

Если два места в приложении могут вызвать обработку платежей одновременно, ты хочешь:

- никаких гонок
- никаких параллельных транзакций

- строгий порядок операций

Один поток + очередь задач даёт это автоматически.

---

5) Как создаётся кастомный dispatcher (механика)

Упрощённо:

1. newSingleThreadExecutor() создаёт ExecutorService:
   - поднимает 1 поток (или поднимет при первой задаче)
   - принимает задачи в очередь
2. asCoroutineDispatcher() создаёт объект Dispatcher, который:
   - при каждом “переключении контекста” (withContext, launch(dispatcher))
   - кладёт continuation (продолжение корутины) как задачу в executor

Именно так корутины “встраиваются” в Java executor.

---

6) Важный момент: здесь есть риск утечки ресурса

Executor создаёт поток, а поток — это ресурс.
Если его не закрыть, он может:

- жить дольше, чем ViewModel
- держать приложение/процесс
- копить “лишние” потоки, если так делают часто

✅ Правильное правило:

Любой dispatcher, созданный из Executor, нужно закрывать (close()),
иначе возможна утечка.

---

7) Как сделать “правильно” в реальном проекте

Ссылка на видео этого руководства на сайте: borisproit.expert

Вариант A (лучший): хранить dispatcher как поле и закрывать в onCleared()

Когда реально нужен single-thread dispatcher

✅ Хорошие случаи:

- обработка платежей/заказов строго по очереди
- запись в файл/лог одним потоком
- доступ к небезопасному SDK, который требует один поток
- сериализация операций с кешем/хранилищем, если Mutex/actor не используешь

❌ Плохие случаи:

- “просто чтобы было не на Main”
- тяжёлые CPU вычисления (лучше Default)
- массовые сетевые запросы (лучше IO)

75. Что такое thread-hopping в корутинах?

```
viewModelScope.launch {
    loadFromNetwork()              // Main → suspends
    withContext(Dispatchers.IO) {
        saveToDb()                 // IO thread
    }
    updateUi()                     // back to Main
}
```

Thread-hopping — это переключение корутины между потоками во время выполнения.
Корутина может начать работу в одном потоке, приостановиться, а затем возобновиться в другом — это нормально и управляется dispatcher’ами и suspend-функциями.

Ссылка на видео этого руководства на сайте: borisproit.expert

Thread-hopping is when a coroutine switches threads during execution.
A coroutine may start on one thread, suspend, and resume on another, depending on dispatchers and suspending functions.

O. Под капотом

76. Что такое Continuation?

```
suspend fun paySuspend(amount: Int): String =
    suspendCoroutine { cont ->
        sdk.pay(amount) { result ->
            result
                .onSuccess { cont.resume(it) }
                .onFailure { cont.resumeWithException(it) }
        }
    }
```

Continuation — это низкоуровневый механизм, с помощью которого Kotlin реализует приостановку и возобновление корутин. Обычно разработчик с ним не работает напрямую.

Continuation — это объект, который хранит состояние приостановленной корутины и описывает, как продолжить её выполнение.
Именно Continuation позволяет корутине остановиться на suspend-точке и позже продолжить с того же места, даже на другом потоке.

Проще:
Continuation = “закладка” в коде + инструкция, как продолжить выполнение.

Continuation is a low-level mechanism that enables coroutine suspension and resumption. Developers usually don’t interact with it directly.

Continuation is an object that stores the state of a suspended coroutine and defines how execution should resume.
It allows a coroutine to pause at a suspend point and later continue from the same place, possibly on a different thread.

Ссылка на видео этого руководства на сайте: borisproit.expert

Simply:
Continuation = execution state + resume logic.

77. Как устроена реализация корутин на уровне байткода?

На JVM корутины реализуются трансформацией компилятором: suspend-функция превращается в машину состояний (state machine) и получает скрытый параметр Continuation.

Когда выполнение доходит до точки suspend, функция возвращает маркер COROUTINE_SUSPENDED, поток освобождается, а продолжение позже происходит через resumeWith(...).

On the JVM, coroutines are implemented via a compiler transformation: a suspend function becomes a state machine and gets a hidden Continuation parameter.

At a suspend point it returns COROUTINE_SUSPENDED, releases the thread, and later continues via resumeWith(...).

Ссылка на видео этого руководства на сайте: borisproit.expert

78. Как coroutine-state машина хранит состояние?

```
class LoadRoomNameContinuation(
    completion: Continuation<String>
) : ContinuationImpl(completion) {

    var label: Int = 0          // execution step
    var room: Any? = null       // saved local variable
}

fun loadRoomName(cont: LoadRoomNameContinuation): Any {
    when (cont.label) {
        0 -> {
            cont.label = 1
            val result = api.getRoom(cont)
            if (result === COROUTINE_SUSPENDED) return COROUTINE_SUSPENDED
            cont.room = result
        }
        1 -> {
            // resumed after getRoom()
        }
    }

    val room = cont.room as Room
    val price = pricing.calc(room)
    return "${room.name}: $price"
}
```

Coroutine state machine хранит состояние в объекте Continuation, который создаёт компилятор.

В этом объекте сохраняются:

Ссылка на видео этого руководства на сайте: borisproit.expert

- текущий шаг выполнения (label)
- локальные переменные suspend-функции
- результат последней suspend-операции

Это позволяет корутине приостановиться и продолжить с нужного места.

The coroutine state machine stores its state inside a Continuation object generated by the compiler.
This object holds:

- the current execution step (label)
- local variables of the suspend function
- the result of the last suspension

This allows the coroutine to pause and later resume from the correct point.

79. Что происходит при suspension?

При suspension корутина временно прекращает выполнение, освобождает поток, а её текущее состояние сохраняется в Continuation.
Корутина не завершена и не отменена — она ждёт возобновления (resume).

During suspension, a coroutine pauses execution, releases the thread, and saves its current state inside the Continuation.
The coroutine is not finished or cancelled — it is waiting to be resumed.

80. Как корутины переключают контексты?

Ссылка на видео этого руководства на сайте: borisproit.expert

```
viewModelScope.launch {
    loadFromNetwork()                   // Main
    withContext(Dispatchers.IO) {
        saveToDb()                      // IO
    }
    updateUi()                          // back to Main
}
```

Здесь корутина сохраняет состояние, приостанавливается и продолжается в нужном контексте без блокировки потока.

Корутины переключают контекст через Dispatcher, используя suspension и resumption.
При withContext или suspend-точке текущий контекст сохраняется, корутина приостанавливается, а затем возобновляется в новом контексте (другом потоке или пуле).

Coroutines switch contexts via a Dispatcher using suspension and resumption.
On withContext or a suspend point, the current context is saved, the coroutine suspends, and then resumes in the new context (different thread or thread pool).

P. ДОПОЛНИТЕЛЬНЫЕ ВОПРОСЫ:

Channels & Actors

81. Что такое Channel?

Ссылка на видео этого руководства на сайте: borisproit.expert

```
val channel = Channel<String>()

viewModelScope.launch {
    channel.send("Room booked")
}

lifecycleScope.launch {
    val event = channel.receive()
    showToast(event)
}
```

Этот код передаёт одноразовое событие из ViewModel в UI в строгом порядке без блокировок.

Channel — это примитив корутин для передачи данных между корутинами в строгом порядке по модели *producer → consumer*.
Он работает без Mutex и без shared mutable state, используется как очередь задач и как альтернатива callback'ам.
Channel — hot-источник: значения отправляются независимо от получателя.

Channel is a coroutine primitive for ordered communication between coroutines using the *producer → consumer* model.
It works without Mutex or shared mutable state, acts as a task queue, and can replace callbacks.
A Channel is hot: values are sent regardless of receivers.

82. Чем отличаются SendChannel и ReceiveChannel?

```
val channel = Channel<String>()

fun sendEvents(): SendChannel<String> = channel
fun receiveEvents(): ReceiveChannel<String> = channel
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Этот код разделяет доступ: один компонент может только отправлять события, другой — только получать.

SendChannel и ReceiveChannel — это разделённые интерфейсы Channel.
SendChannel позволяет только отправлять данные,
ReceiveChannel — только получать данные.
Это используется для безопасной архитектуры, чтобы одна часть кода не могла делать лишние операции.

SendChannel and ReceiveChannel are separate interfaces of Channel.
SendChannel allows sending only,
ReceiveChannel allows receiving only.
This improves safety and encapsulation in architecture.

83. Что такое actor модель?

Ссылка на видео этого руководства на сайте: borisproit.expert

```
val bookingActor = viewModelScope.actor<BookingMsg> {
    var bookedRooms = 0

    for (msg in channel) {
        when (msg) {
            is BookingMsg.Book -> bookedRooms++
            is BookingMsg.GetCount -> msg.reply.complete(bookedRooms)
        }
    }
}

// Send commands
viewModelScope.launch { bookingActor.send(BookingMsg.Book) }

// Ask for current state (request-reply)
viewModelScope.launch {
    val reply = CompletableDeferred<Int>()
    bookingActor.send(BookingMsg.GetCount(reply))
    val count = reply.await()
    showBookedRooms(count)
}
```

Этот код создаёт bookingActor, который хранит bookedRooms только внутри себя и обрабатывает команды бронирования по очереди, поэтому состояние меняется безопасно без блокировок.

actor — это модель конкурентности, где состояние изолировано внутри одной корутины, а доступ к нему идёт только через сообщения.

В Kotlin coroutines actor реализован как корутина + Channel: он читает сообщения из своего канала строго по очереди, поэтому можно безопасно менять состояние без Mutex и без shared mutable state.

actor берётся из kotlinx.coroutines и создаётся внутри CoroutineScope через builder actor { ... }.

Ссылка на видео этого руководства на сайте: borisproit.expert

An actor is a concurrency model where state is isolated inside a single coroutine and can be accessed only via messages.
In Kotlin coroutines, an actor is essentially a coroutine + a Channel: it processes messages sequentially, so you can safely mutate state without locks or shared mutable state.
actor comes from kotlinx.coroutines and is created inside a CoroutineScope using the actor { ... } builder.

84. Когда actor лучше Mutex?

```
sealed class BookingMsg {
    object Book : BookingMsg()
    data class GetCount(val reply: CompletableDeferred<Int>) : BookingMsg()
}

val bookingActor = viewModelScope.actor<BookingMsg> {
    var bookedRooms = 0
    for (msg in channel) when (msg) {
        is BookingMsg.Book -> bookedRooms++
        is BookingMsg.GetCount -> msg.reply.complete(bookedRooms)
    }
}
```

Этот actor безопасно держит счётчик бронирований и обрабатывает все команды строго по очереди, поэтому никакой Mutex не нужен.

actor лучше Mutex, когда у тебя есть общий кусок состояния, и к нему идут частые операции (команды/ивенты), и ты хочешь обрабатывать их строго по очереди как через очередь сообщений. Тогда состояние живёт в одном месте, нет shared mutable state и почти не нужен ручной контроль блокировок.
Mutex лучше, когда нужно кратко защитить маленький критический участок вокруг уже существующей переменной, без построения очереди и протокола сообщений.

Когда выбирать actor:

- много конкурентных запросов к одному состоянию (например, бронирования, баланс, инвентарь)
- нужен строгий порядок операций (FIFO)

Ссылка на видео этого руководства на сайте: borisproit.expert

- хочется “командную” модель: send/обработать/ответить
- нужна изоляция логики и состояния (один владелец состояния)

Use an actor over a Mutex when you have shared state with many concurrent operations and you want sequential, ordered processing via a message queue. The state lives in one place, no shared mutable state, minimal locking complexity.
Use a Mutex when you just need to protect a small critical section around an existing variable without introducing message protocols.

Choose actor when:

many concurrent operations hit the same state (bookings, balance, inventory)

strict ordering matters (FIFO)

command/event style processing fits (send → handle → reply)

you want single-owner state isolation

---

86. Что такое hot flow?

Ссылка на видео этого руководства на сайте: borisproit.expert

```
val state = MutableStateFlow(0)

viewModelScope.launch {
    state.value = 1
    state.value = 2
}

lifecycleScope.launch {
    state.collect { value ->
        show(value)
    }
}
```

Этот код показывает hot flow: значения эмитятся независимо от подписки, а коллектор получает актуальные обновления состояния.

Hot flow — это поток данных, который эмитит значения независимо от подписчиков.
Он живёт сам по себе, хранит или рассылает данные всем коллекторам, а новые подписчики получают текущие/последующие значения, а не запускают поток заново.
Примеры: StateFlow, SharedFlow.

A hot flow emits values independently of collectors.
It exists on its own, shares emissions among collectors, and new collectors receive current/ongoing values rather than starting a new execution.
Examples: StateFlow, SharedFlow.

87. Что делать при медленном collector?

Ссылка на видео этого руководства на сайте: borisproit.expert

```
roomsFlow
    .conflate()
    .collectLatest { rooms ->
        renderRooms(rooms) // slow UI work
    }
```

Этот код защищает от медленного collector:

conflate() выбрасывает промежуточные обновления, а collectLatest() отменяет старую отрисовку и всегда показывает самое актуальное.

Если collector обрабатывает данные медленно, возникает backpressure: эмиттер производит быстрее, чем потребитель успевает. Решения зависят от задачи:

buffer() — дать запас буфера, чтобы не тормозить producer

conflate() — пропускать промежуточные значения, оставляя только последнее

collectLatest() — отменять обработку предыдущего значения и обрабатывать только самое свежее

sample() / debounce() — уменьшать частоту событий (например, UI)

flowOn(...) / withContext(...) — перенести тяжёлую обработку с Main на фон

SharedFlow/Channel — настроить буфер (extraBufferCapacity) и стратегию переполнения (BufferOverflow)

оптимизировать обработку внутри collect (не делать тяжёлую работу на Main)

A slow collector causes backpressure: producer emits faster than consumer can process. Common fixes:

buffer() — add buffering so producer isn't blocked

conflate() — drop intermediate values, keep only the latest

collectLatest() — cancel previous work, process only newest

sample() / debounce() — reduce emission rate (often for UI)

flowOn(...) / withContext(...) — move heavy work off Main

configure SharedFlow/Channel buffering and overflow strategy

optimize the work inside collect

88. Как Flow ведёт себя при отмене?

Flow реагирует на отмену автоматически.
Когда корутина-collector отменяется (lifecycle, ViewModel, job.cancel), сбор Flow немедленно прекращается, upstream тоже отменяется, ресурсы освобождаются.
Никаких дополнительных действий обычно не нужно — отмена кооперативная и безопасная.

Flow is cancellation-aware by default.
When the collecting coroutine is cancelled (lifecycle, ViewModel, job.cancel), collection stops immediately, upstream is cancelled as well, and resources are released.
In most cases, no extra handling is required.

Ссылка на видео этого руководства на сайте: borisproit.expert

89. Как Flow связан с structured concurrency?

```
viewModelScope.launch {
    roomsFlow.collect { rooms ->
        _state.value = rooms
    }
}
```

Этот код показывает, что сбор Flow привязан к viewModelScope: когда ViewModel очищается, сбор автоматически отменяется и никаких фоновых задач не остаётся.

Flow полностью подчиняется structured concurrency.
Он всегда собирается внутри CoroutineScope, наследует его Job и автоматически отменяется вместе с родителем.
Это гарантирует, что сбор данных не переживёт жизненный цикл экрана или ViewModel и не приведёт к утечкам.

Flow fully follows structured concurrency.
It is always collected inside a CoroutineScope, inherits its Job, and is automatically cancelled with its parent.
This ensures that flow collection never outlives its scope and prevents leaks.

---

91. Что такое Selector API?

Ссылка на видео этого руководства на сайте: borisproit.expert

```
val bookingChannel = Channel<String>()
val cancelChannel = Channel<Unit>()

viewModelScope.launch {
    select<Unit> {
        bookingChannel.onReceive { booking ->
            handleBooking(booking)
        }
        cancelChannel.onReceive {
            handleCancel()
        }
    }
}
```

Этот код ждёт сразу два события (бронирование или отмену) и выполняет обработку того, которое пришло первым.

Selector API — это механизм корутин, который позволяет ожидать сразу несколько suspend-событий и реагировать на то, которое произошло первым.
Он используется, когда нужно выбрать между несколькими источниками: Channel, Deferred, Job и т.д., без блокировок и без polling.

Selector API is a coroutine mechanism that lets you wait for multiple suspendable events and react to the one that happens first.
It is used to choose between multiple sources like Channel, Deferred, or Job without blocking or polling.

Ссылка на видео этого руководства на сайте: borisproit.expert

92. Как CoroutineContext хранит элементы?

```
val context =
    Job() +
    Dispatchers.IO +
    CoroutineName("SyncPrices")

println(context[Job])
println(context[CoroutineName])
```

Этот код показывает, что CoroutineContext хранит элементы по ключу типа и позволяет доставать любой элемент независимо, как из словаря.

CoroutineContext — это набор элементов, хранящийся как ассоциативная структура (map-подобная), где каждый тип элемента уникален.
Элементы (Job, Dispatcher, Name и т.д.) объединяются через оператор +.
Если добавить элемент с тем же ключом, старый будет заменён.

CoroutineContext is a set of elements stored in a map-like structure, where each element type is unique.
Elements (Job, Dispatcher, Name, etc.) are combined using the + operator.
If a new element with the same key is added, it replaces the old one.

93. Что такое ThreadLocal + coroutines?

```
val requestId = ThreadLocal<String>()

viewModelScope.launch(
    Dispatchers.IO + requestId.asContextElement("REQ-123")
) {
    log(requestId.get()) // "REQ-123"
}
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Этот код показывает, что значение ThreadLocal сохраняется и корректно передаётся внутри корутины, даже если она переключается между потоками.

ThreadLocal — это хранилище данных, привязанное к потоку, а корутины могут переключаться между потоками, поэтому обычный ThreadLocal не работает корректно с корутинами.

Для этого в корутинах используется специальная интеграция — ThreadLocal.asContextElement(), которая сохраняет и восстанавливает значение ThreadLocal при переключении контекста.

ThreadLocal stores data per thread, but coroutines can hop between threads, so plain ThreadLocal does not work correctly with coroutines.

Kotlin provides ThreadLocal.asContextElement() to propagate and restore ThreadLocal values across coroutine suspensions and context switches.

94. Как управлять coroutine-debugging?

```
val scope = CoroutineScope(
    Dispatchers.IO + CoroutineName("PriceSync")
)

scope.launch {
    delay(1000)
}
```


```
val scope = CoroutineScope(
    Dispatchers.IO + CoroutineName("PriceSync")
)

scope.launch {
    delay(1000)
}
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Этот код даёт корутине имя и при включённом debug-режиме позволяет видеть её в дебаггере и логах с понятным контекстом.

Coroutine debugging позволяет видеть корутины, их имена, статусы и стек вызовов.
Управляется через JVM-флаг, CoroutineName, и debug-интеграцию IDE. Это помогает понимать, где корутина запущена, почему она зависла или была отменена.

Coroutine debugging lets you inspect coroutines, their names, states, and stack traces.
It is controlled via JVM flags, CoroutineName, and IDE debug support, helping diagnose hangs, cancellations, and execution flow.

95. Как устроен EventLoop в корутинах?

```
runBlocking {
    launch {
        println("A")
        delay(100)
        println("B")
    }

    println("C")
}
```

EventLoop в корутинах — это механизм, который исполняет корутины на одном потоке, ставя их задачи в очередь и выполняя по очереди.
Он нужен для неблокирующего выполнения, обработки delay, yield, withContext, и для работы Dispatchers.Unconfined и runBlocking.

The coroutine EventLoop is a mechanism that runs coroutines on a single thread using a task queue.
It enables non-blocking execution, handles delay, yield, withContext, and is used by Dispatchers.Unconfined and runBlocking.

Ссылка на видео этого руководства на сайте: borisproit.expert

96. Как корутины взаимодействуют с JNI?

Корутины не имеют прямого взаимодействия с JNI.
Для JNI корутина — это обычный код, выполняющийся в обычном потоке.
Важно помнить: JNI-вызовы блокирующие, поэтому их нельзя выполнять на Dispatchers.Main — только на Dispatchers.IO или кастомном dispatcher'е.

Coroutines have no special integration with JNI.
From JNI's perspective, a coroutine is just regular code running on a thread.
JNI calls are blocking, so they must run on Dispatchers.IO or a custom dispatcher, never on Dispatchers.Main.

97. Как корутины ведут себя при ANR?

Корутины сами по себе не спасают от ANR. ANR происходит, когда Main thread слишком долго занят (обычно ~5 секунд) и не обрабатывает события.
Если корутина выполняет долгую/блокирующую работу на Dispatchers.Main (или ты вызвал блокирующий код внутри UI-корутины), ты получишь ANR так же, как без корутин.
Отмена корутины тоже не "вылечит" ANR, если Main уже заблокирован.

Coroutines do not prevent ANR by themselves. ANR happens when the Main thread is blocked too long (typically ~5s) and can't process events.
If a coroutine runs long or blocking work on Dispatchers.Main (or calls blocking code from a UI coroutine), you can still get ANR.
Cancelling a coroutine won't fix ANR if the Main thread is already blocked.

98. Как проектировать корутин-архитектуру в больших проектах?

Ссылка на видео этого руководства на сайте: borisproit.expert

```
interface AppDispatchers {
    val Main: CoroutineDispatcher
    val IO: CoroutineDispatcher
    val Default: CoroutineDispatcher
}

class RoomRepository(
    private val api: RoomApi,
    private val db: RoomDao,
    private val dispatchers: AppDispatchers
) {
    suspend fun loadRooms(): List<Room> = withContext(dispatchers.IO) {
        val rooms = api.getRooms()
        db.saveRooms(rooms)
        rooms
    }
}

class RoomsViewModel(
    private val repo: RoomRepository
) : ViewModel() {

    fun load() = viewModelScope.launch {
        val rooms = repo.loadRooms()
        render(rooms)
    }
}
```

Этот код показывает базовую архитектуру: UI запускает корутину в viewModelScope, репозиторий делает I/O через withContext(IO), а dispatcher'ы приходят через DI — удобно тестировать и масштабировать.

Основа архитектуры: structured concurrency + явные scope'ы + DI для dispatcher'ов + единый подход к ошибкам и отмене.
Практически это значит:

- UI запускает корутины только для UI-задач (viewModelScope/lifecycleScope)
- бизнес-логика и I/O — в suspend/Flow в репозиториях/юзкейсах

Ссылка на видео этого руководства на сайте: borisproit.expert

- dispatcher’ы и scope’ы не хардкодятся, а внедряются (DI)
- отмена и ошибки предсказуемы: SupervisorJob там, где нужно “не валить всё”, и обычный Job там, где нужно “всё или ничего”

The foundation is structured concurrency + explicit scopes + DI for dispatchers + consistent cancellation/error strategy.
In practice:

- UI launches coroutines only for UI concerns (viewModelScope/lifecycleScope)
- business/I/O work lives in suspend/Flow in repositories/use-cases
- dispatchers/scopes are injected, not hardcoded
- cancellation & errors are predictable: SupervisorJob where isolation is needed, Job where all-or-nothing is required

99. Какие anti-patterns есть при работе с корутинами?

Самые частые anti-patterns в больших проектах:

- запуск “вечных” корутин без владельца жизненного цикла (GlobalScope, CoroutineScope() без cancel)
- блокирующие вызовы в Dispatchers.Main (сеть/JNI/БД/Thread.sleep)
- “проглатывание” CancellationException
- async без await и launch там, где нужен результат (путают семантику)
- создание новых CoroutineScope в каждом классе без стратегии отмены
- withContext(IO) внутри ViewModel повсюду вместо нормального слоя репозитория/юзкейса
- shared mutable state без защиты (гонки)
- тяжёлая работа внутри collect на Main, отсутствие backpressure-операторов

Ссылка на видео этого руководства на сайте: borisproit.expert

- “try/catch на всё” вместо понятной стратегии ошибок (и неверное ожидание, что handler спасёт всё)

Common coroutine anti-patterns:

- unstructured “forever” coroutines (GlobalScope, unmanaged CoroutineScope() without cancel)
- blocking work on Dispatchers.Main (network/JNI/DB/Thread.sleep)
- swallowing CancellationException
- async without await, and using launch when a result is needed
- creating scopes everywhere with no cancellation strategy
- scattering withContext(IO) in UI instead of proper repository/use-case layering
- unsafe shared mutable state (race conditions)
- heavy work in collect on Main, missing backpressure operators
- catching everything blindly instead of a consistent error strategy