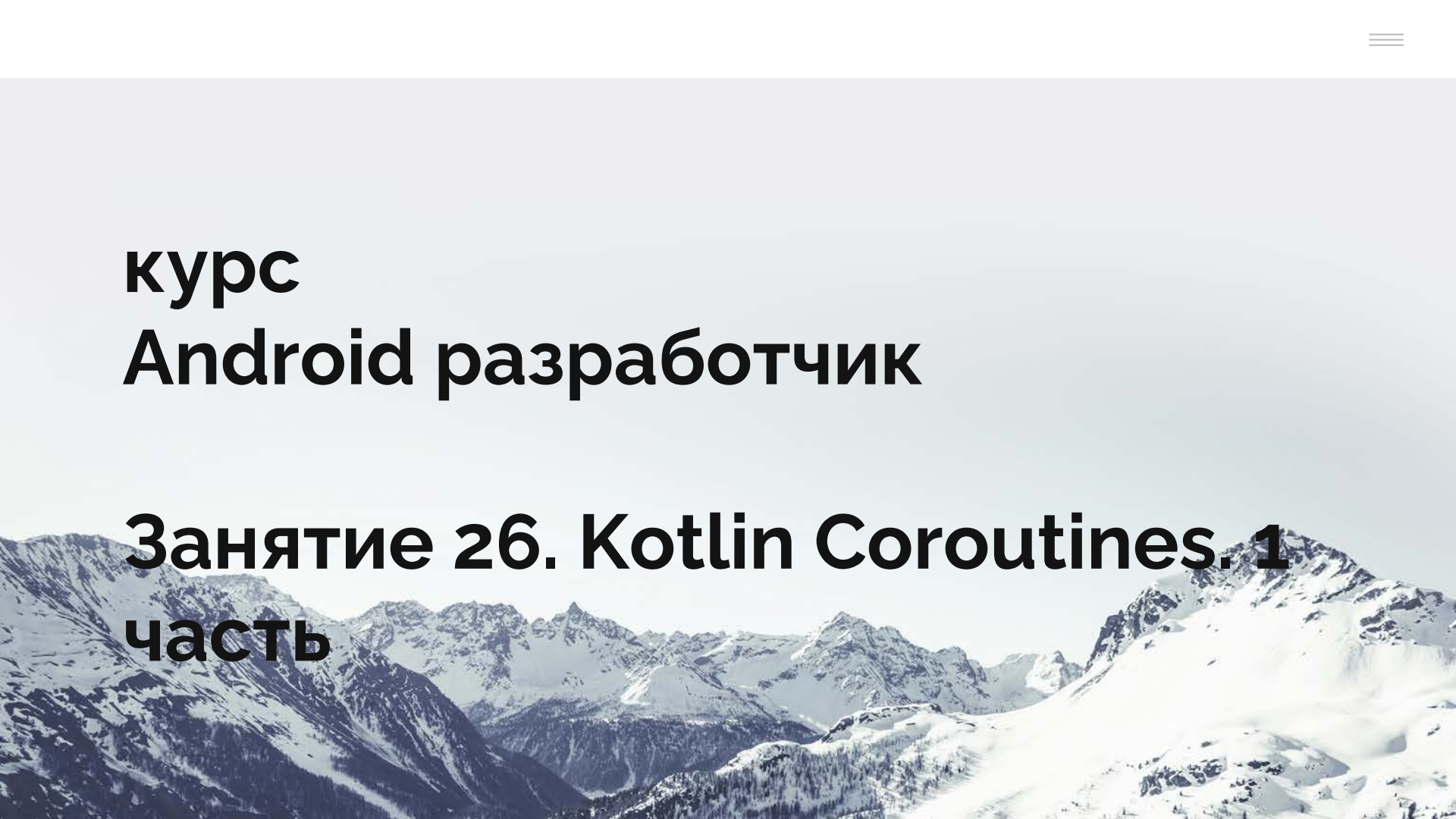
<TeachMeSkills/>



курс
Android разработчик

Занятие 26. Kotlin Coroutines. 1 часть

Агенда занятия

Введение в многопоточность

Понятие корутины

suspend-функция

Запуск корутины

Базовая терминология корутин

- Введение в многопоточность
- Понятие корутины
- suspend-функция
- Запуск корутины
- Базовая терминология корутин

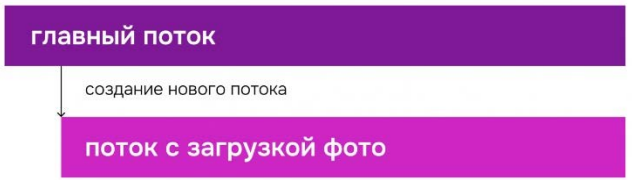
Main Thread или UI Thread в Android

В Android (и вообще в программировании), поток — это самостоятельная единица выполнения кода. По умолчанию весь ваш код в Activity и Fragment работает в главном потоке (**Main Thread или UI Thread**).
Если перегрузить этот поток длительной задачей (например, загрузкой файла, медленным вычислением), пользовательский интерфейс перестаёт реагировать: кнопки не нажимаются, анимации замирают, появляется "белый экран".
Android следит за этим и если UI-поток заблокирован более 5 секунд, выбрасывает ошибку **ANR (Application Not Responding)** — приложение закрывается.

Как избежать ANR?

Вариант 1:

Сделать сетевой запрос в отдельном потоке, тем самым выполняя его параллельно



Проблема создания потока?

Создание потоков далеко небесплатная штука с точки зрения производительности, и потому создание любого нового потока - это накладные ресурсы для вашего программы.
Да, иногда без потока не обойтись, когда речь идёт о тяжёлых вычислениях, и они реально сильно ускорят вашу программу, сделав затраты на создание нового потока незначительными. Но в случае сетевого запроса это может быть избыточным, так как такой сетевой запрос не несёт большой нагрузки на процессор, ведь все время ожидания приходится на отправку вашего запроса туда и обратно, где устройство пользователя просто ждёт ответ, вообще ничего не делая.

Введение в многопоточность

Понятие корутины

suspend-функция

Запуск корутины

Базовая терминология корутин

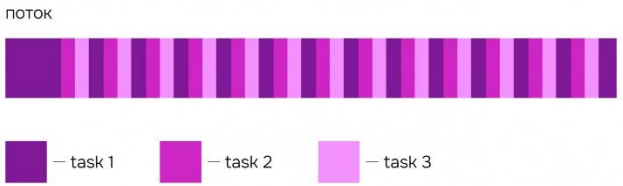
Альтернативный вариант – асинхронность

Асинхронность - это парадигма, которая позволяет выполнять и обрабатывать операции без ожидания их завершения. То есть вместо последовательного выполнения асинхронность позволяет коду продолжать выполняться, пока некоторые задачи выполняются в фоновом режиме, и обрабатывать их результаты по мере их готовности.

Это достигается за счет неблокирующих операций, которые не приостанавливают поток, а продолжают выполн об этом через механизм колбэков.

Асинхронность != многопоточность

Асинхронность не блокирует основной поток, а как бы нарезает его кусочками, что создаёт иллюзию перед пользователем параллельного выполнения кода, хотя по настоящему параллельный код не создаётся (в отличие от потоков). Причем, если операций много и они ничего не ожидают (как в случае с ответом от сервера), то асинхронность будет достигаться за счет очень быстрог



Что такое корутины в Kotlin?

Корутины - легковесные асинхронные конструкции, которые позволяют выполнять операции параллельно и эффективно управлять асинхронным кодом. Они являются частью языка Kotlin и позволяют писать асинхронный код, который выглядит и ведёт себя как синхронный, что упрощает чтение и поддержку кода.

Введение в многопоточность

Понятие корутины

suspend-функция

Запуск корутины

Базовая терминология корутин

Suspend функции

Для того, чтобы сказать нашей программе, что не нужно блокировать поток, дожидаясь ответа от определенного метода, используется ключевое слово suspend. Этим словом помечается тот метод, который как раз и способен заблокировать наш поток на длительное время (например, таким примером может послужить тот самый метод сетевого вызова).

Введение в многопоточность

Понятие корутины

suspend-функция

▶ **Запуск корутины**

Scopes, withContext

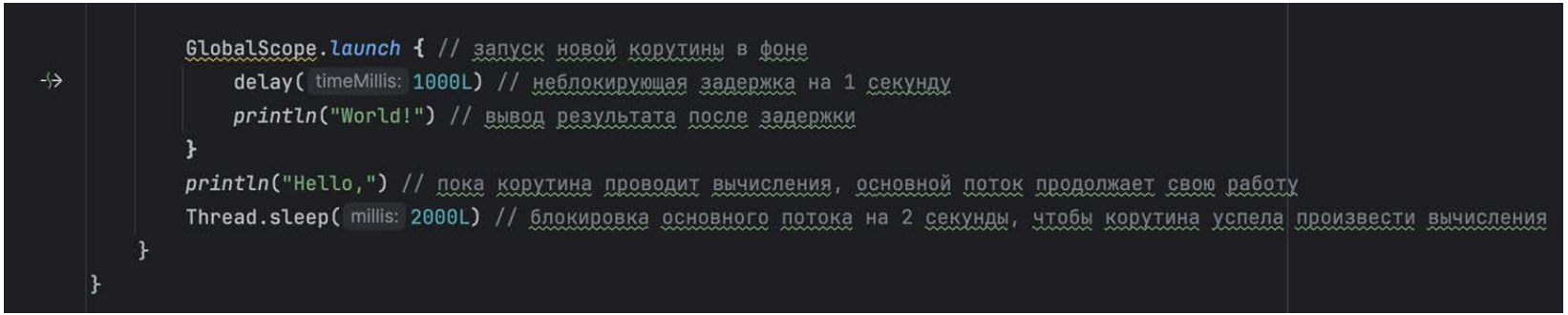
Job

Async/await; launch/join

Запуск корутины

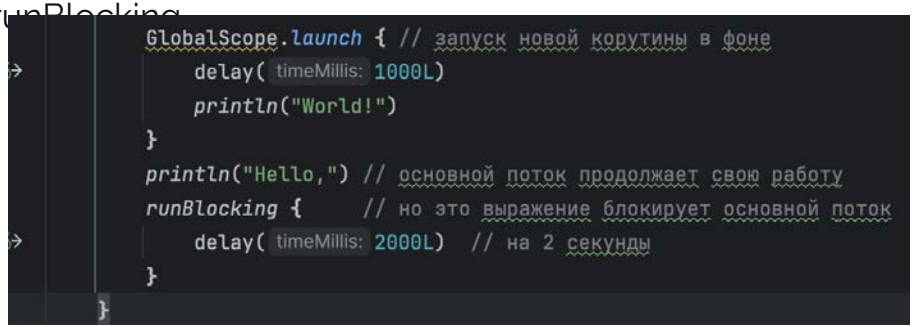
Корутины запускаются с помощью конструктора корутин в контексте какого-то CoroutineScope.

Здесь мы запускаем новую корутину в GlobalScope, что означает, что время жизни новой



Запуск корутины

В первом примере смешаны две
функции: неблокирующая delay() и блокирующая Thread.sleep(). Легко забыть, какая из них блокирует основной поток, а какая нет. Давайте подробно рассмотрим блокировку с помощью билдера runBlocking



Введение в многопоточность

Понятие корутины

suspend-функция

Запуск корутины

- Базовая терминология корутин

Scope

Отслеживает любую корутину, которую создает, используя launch или async. Scope хранит все ссылки на корутины, запущенные в нем. Scope может отменить выполнение всех дочерних корутин, если возникнет ошибка или операция будет отменена.

- Отмена дочернего scope приведет к отмене родительского scope и наоборот.
- Scope будет завершен успешно, когда выполнятся все корутины в нем.

Основные scope

lifecycleScope - Расширение для корутин, привязанное к жизненному циклу компонента, такого как Activity или Fragment. Он автоматически управляет запуском и отменой корутин в зависимости от состояния жизненного цикла компонента, что помогает избежать утечек памяти и ошибок, связанных с обращением к уничтоженным компонентам.

viewModelScope - Специальный диспетчер, который связан с жизненным циклом ViewModel. Он позволяет запускать корутины, которые будут автоматически отменены, когда ViewModel будет уничтожена. Это делает viewModelScope отличным выбором для выполнения асинхронных операций, связанных с обработкой данных в ViewModel.

CoroutineContext

Определяет поведение корутины. Является набором параметров для выполнения корутин.
• Каждая корутина выполняется в каком-либо контексте.
• Явно не создается, задается в scope либо при запуске корутины (в launch).
• Можно объединить несколько контекстов в один.

withContext
Переносит выполнение текущей корутины на новый контекст, в большинстве случаев на новый диспетчер.

Билдеры корутин

• Launch - функция, запускающая новую корутину в заданном контексте, которая не блокирует текущий поток и возвращает объект Job, позволяющий управлять выполнением корутины. Запустил и забыл.

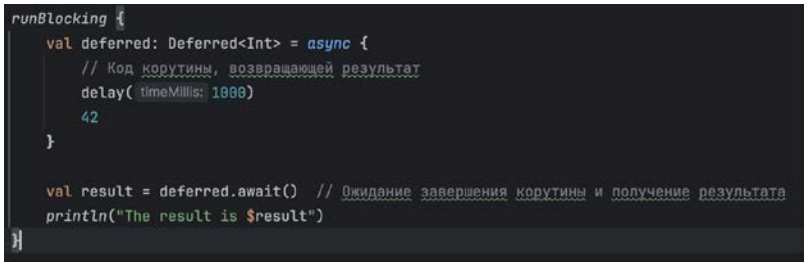
• Async - функция, которая запускает новую корутину и возвращает объект Deferred, позволяющий получить результат выполнения корутины в будущем. Позволяет использовать функцию await() для получения результата, который будет доступен после завершения корутины.

AwaitAll() - Позволяет параллельно ожидать завершения нескольких корутин и собрать их результаты. Она используется для того, чтобы запустить несколько асинхронных операций и дождаться их всех одновременно.

Билдеры корутин

```
runBlocking {
    val job = launch {
        // Код корутины
        delay( timeMillis: 1000)
        println("Hello from coroutine!")
    }

    job.join()  // Ожидание завершения корутины
}
```



Dispatchers

Dispatcher определяет, на каком потоке или пуле потоков будет выполняться код корутины. Он управляет распределением задач между потоками и позволяет выбирать подходящий контекст для выполнения операций.

- Dispatcher.Default
- Dispatcher.IO
- Dispatcher.Main
- Dispatcher.Unconfined

Dispatcher.Main

Специальный диспетчер, который использует главный поток Android (или основной поток в других платформах). Он предназначен для выполнения задач, связанных с пользовательским интерфейсом, таких как обновление UI элементов, обработка событий и другие задачи, которые должны быть выполнены в главном потоке.

Dispatcher.IO

Используется для выполнения блокирующих операций ввода-вывода, таких как работа с файлами или сетевые запросы. Он оптимизирован для работы с большим количеством фоновых задач, распределяя их по пулу потоков.

• Пул потоков имеет динамический размер. Количество потоков может увеличиваться в зависимости от текущей нагрузки и количества ожидающих задач.

• Если все потоки заняты, новые задачи помещаются в очередь и обрабатываются по мере освобождения потоков.

• Под капотом CachedThreadPool

Dispatcher.Default

Предназначен для выполнения задач, требующих значительных вычислительных ресурсов, таких как обработка данных или выполнение сложных алгоритмов. Он использует пул потоков, оптимизированный для CPU-ориентированных задач.

• Использует пул потоков, размер которого обычно соответствует числу доступных процессоров (или ядер).

• Не рекомендуется для I/O операций так как использует пул с ограниченным числом потоков, блокировка одного из этих потоков может снизить общую производительность приложения, особенно если таких операций много.

• Используется по умолчанию в launch и async.

• Под капотом FixedThreadPool

Dispatcher.Unconfined

Запускает корутину в текущем потоке, но не привязан к какому-либо конкретному потоку. Это означает, что корутина может продолжить выполнение в другом потоке после первой приостановки.

Dispatchers

Смотрим проект.

Job

Основной механизм управления корутинами в Kotlin, который позволяет эффективно управлять жизненным циклом корутин, их отменой и композицией.

• Когда вы запускаете корутину с помощью функций launch или async, возвращается объект Job, который представляет корутину.

• На основе Job можно организовать иерархию parent-child.

Базовая терминология корутин

Coroutine — сама "ниточка" выполнения.

suspend-функция — функция, которую можно приостановить (delay, сетевой запрос).

Job — объект управления корутиной (можно отменить, узнать статус).

Scope — область жизни корутин (например, lifecycleScope, viewModelScope).

Dispatcher — где выполняется корутина (Main, IO, Default).

Билдеры(launch, async, runBlocking) –расширения, запускающие корутину

Связь:

Scope запускает корутину (Coroutine), которая может быть suspend-функцией, выполняться на определённом Dispatcher и быть представлена Job.

Базовая терминология корутин

Смотрим проект.

Практика

Задачи

Задача 1: Запустите корутину, обновляющую textView на экране

Так же отобразите кнопку меняющую другой textView. При запуске корутины она не должна блокироваться.

Задачи

Задача 2: Список из 1000 чисел

Создайте функцию которая генерирует список из 1000 случайных чисел, считает их сумму и показывает результат пользователю.

Задачи

Задача 3: Отмена задачи

Долгая загрузка (например, счетчик от 10 до 0, каждую секунду) — если пользователь нажимает "Отмена", корутина прерывается, в UI показывается "Операция отменена".

Задачи

Задача 3: две корутины

Две async-корутины: первая имитирует сетевой запрос (delay 700), вторая — работу с файловой системой (delay 1200), после завершения обеих показывается комбинированный результат.

Q&A

Домашнее задание

Задачи

Задача 1: Экран "Асинхронный запрос":

Экран "Асинхронный запрос":

- Кнопка "Загрузить данные"

- ProgressBar (показывается во время загрузки, скрывается после)

- TextView для результата или ошибки

- Используйте suspend-функцию с delay (и случайно выбрасывайте Exception для имитации ошибки)

- Всё реализуйте через корутины, переключайте контексты (IO/Main)

- Отмена загрузки при закрытии экрана (ViewModel + viewModelScope)

- Показывайте результат или ошибку

Бонус:

Добавьте возможность запускать несколько таких "загрузок" параллельно, выводите их статусы в RecyclerView.

Q&A

Ваши вопросы

Спасибо