

Clean architecture

Навигация в многомодульном проекте:

- 1) Добавить модули в апп
- 2) заинклюдить нав графы в общий нав граф этого модуля, учитывать соответствия id — назовения клавиши в меню
- 3) Засетапить нав контроллер :

```
val navHostFragment =  
    supportFragmentManager.findFragmentById(R.id.fragmentContainerView) as NavHostFragment  
val navController = navHostFragment.navController  
binding.bottomNavbar.setupWithNavController(navController)
```

DiffUtil

DiffUtil - дженерик класс в котором нужно переопределить две дженерик функции : `areItemsTheSame` и `areContentsTheSame` что бы понимать как необходимо сравнивать объекты. Процесс нахождения самого лучшего по времени алгоритма происходит змейками по алгоритму Майерса

Koroutine

- `viewModelScope` is a predefined CoroutineScope that is included with the ViewModel KTX extensions. Note that all coroutines must run in a scope. A CoroutineScope manages one or more related coroutines.
- `launch` is a function that creates a coroutine and dispatches the execution of its function body to the corresponding dispatcher.
- `Dispatchers.IO` indicates that this coroutine should be executed on a thread reserved for I/O operations.

Manage long-running tasks

Coroutines build upon regular functions by adding two operations to handle long-running tasks. In addition to `invoke` (or `call`) and `return`, coroutines add `suspend` and `resume`:

- `suspend` pauses the execution of the current coroutine, saving all local variables.
- `resume` continues execution of a suspended coroutine from the place where it was suspended.

You can call `suspend` functions only from other `suspend` functions or by using a coroutine builder such as `launch` to start a new coroutine.

Start a coroutine

You can start coroutines in one of two ways:

- `launch` starts a new coroutine and doesn't return the result to the caller. Any work that is considered "fire and forget" can be started using `launch`.
- `async` starts a new coroutine and allows you to return a result with a `suspend` function called `await`

Parallel decomposition (ВАЖНО обязательно дождаться конца выполнения корутины)

All coroutines that are started inside a `suspend` function must be stopped when that function returns, so you likely need to guarantee that those coroutines finish before returning. With *structured concurrency* in Kotlin, you can define a `CoroutineScope` that starts one or more coroutines. Then, using `await()` (for a single coroutine) or `awaitAll()` (for multiple coroutines), you can guarantee that these coroutines finish before returning from the function.

As an example, let's define a `CoroutineScope` that fetches two documents asynchronously. By calling `await()` on each deferred reference, we guarantee that both `async` operations finish before returning a value:

```
suspend fun fetchTwoDocs() =  
    coroutineScope {  
        val deferredOne = async { fetchDoc(1) }  
        val deferredTwo = async { fetchDoc(2) }  
        deferredOne.await()  
        deferredTwo.await()  
    }
```

Job

A [Job](#) is a handle to a coroutine. Each coroutine that you create with `launch` or `async` returns a `Job` instance that uniquely identifies the coroutine and manages its lifecycle. You can also pass a `Job` to a `CoroutineScope` to further manage its lifecycle, as shown in the following example:

```
...
fun exampleMethod() {
    // Handle to the coroutine, you can control its lifecycle
    val job = scope.launch {
        // New coroutine
    }

    if (...) {
        // Cancel the coroutine started above, this doesn't affect the scope
        // this coroutine was launched in
        job.cancel()
    }
}
```

CoroutineContext

A [CoroutineContext](#) defines the behavior of a coroutine using the following set of elements:

- [Job](#): Controls the lifecycle of the coroutine.
- [CoroutineDispatcher](#): Dispatches work to the appropriate thread.
- [CoroutineName](#): The name of the coroutine, useful for debugging.
- [CoroutineExceptionHandler](#): Handles uncaught exceptions.

For new coroutines created within a scope, a new `Job` instance is assigned to the new coroutine, and the other `CoroutineContext` elements are inherited from the containing scope. You can override the inherited elements by passing a new `CoroutineContext` to the `launch` or `async` function. Note that passing a `Job` to `launch` or `async` has no effect, as a new instance of `Job` is always assigned to a new coroutine.

Test coroutine :

```
@get:Rule
val coroutineScope = MainCoroutineScopeRule()
```

Both Room and Retrofit use a custom dispatcher and do not use Dispatchers.IO.

Room will run coroutines using the default [query](#) and [transaction](#) Executor that's configured.

Retrofit will create a new `Call` object under the hood, and call [enqueue](#) on it to send the request asynchronously.

Test LiveData :

```
@get:Rule  
val instantTaskExecutorRule = InstantTaskExecutorRule()
```

LiveDataObjc.getValueForTest()

Flow

A Flow is an **async sequence of values**

Flow produces values one at a time (instead of all at once) that can generate values from `async` operations like network requests, database calls, or other `async` code. It supports coroutines throughout its API, so you can transform a flow using coroutines as well!

Поток построен с нуля с использованием сопрограмм. Используя механизм приостановки и возобновления сопрограмм, они могут синхронизировать выполнение производителя (поток) с потребителем (сбор).

Если вы использовали реактивные потоки и знакомы с концепцией противодавления, в Flow оно реализовано путем приостановки сопрограммы.

collect

common

```
suspend fun Flow<*>.collect()
```

Terminal flow operator that collects the given flow but ignores all emitted values. If any exception occurs during collect or in the provided flow, this exception is rethrown from this method.

It is a shorthand for `collect {}`.

This operator is usually used with `onEach`, `onCompletion` and `catch` operators to process all emitted values and handle an exception that might occur in the upstream flow or during processing, for example:

```
flow  
.onEach { value -> process(value) }  
.catch { e -> handleException(e) }  
.collect() // trigger collection of the flow
```

Terminal operators (для того что бы положить данные) :

- [toList](#)
- [first](#)
- [single](#)
- `collect`

Executing a Flow is called **collecting** a flow. By default, a Flow will not do anything until it has been **collected** which means applying any **terminal operator**.
myFlow.toList() // toList collects this flow and adds the values to a List

We also say an individual value is **collected** from the Flow by a terminal operator.

```
myFlow.collect { item -> println("$item has been collected") }
```

Flow supports structured concurrency

Because a flow allows you to consume values only with terminal operators, it can support structured concurrency.

When the consumer of a flow is cancelled, the entire Flow is cancelled. Due to structured concurrency, it is impossible to leak a coroutine from an intermediate step

The **asLiveData** operator converts a Flow into a LiveData with a configurable timeout.

Just like the `liveData` builder, the timeout will help the Flow survive restart. If another screen observes before the timeout, the Flow won't be cancelled.

Calling `flowOn` has two important effects on how the code executes:

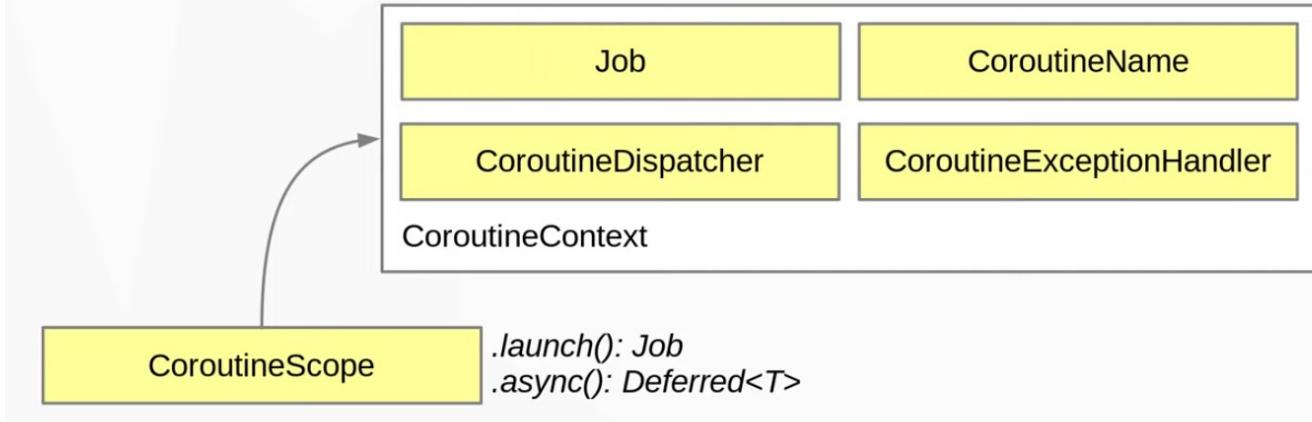
1. Launch a new coroutine on the `defaultDispatcher` (in this case, `Dispatchers.Default`) to run and collect the flow **before** the call to `flowOn`.
2. Introduces a buffer to send results from the new coroutine to later calls.
3. Emit the values from that buffer into the Flow **after** `flowOn`. In this case, that's `asLiveData` in the `ViewModel`

launchIn(viewModelScope) – очень важно так мы определяем скоуп в котором это будет работать

Properety animation

Definition: A **property**, to the animation system, is a field that is exposed via setters and getters, either implicitly (as properties are in Kotlin) or explicitly (via the setter/getter pattern in the Java programming language). There are also a special case of properties exposed via the class `android.util.Property` which is used by the `View` class, which allows a type-safe approach for animations, as you'll see later. The Animator system in Android was specifically written to animate properties, meaning that it can animate anything (not just UI elements) that has a setter (and, in some cases, a getter).

Kotlin Coroutines



- **launch(): Job** просто запускает блок кода асинхронно
 - Можно подождать завершения с помощью метода Job.join()
 - Метод join() – это suspend-метод
- **async(): Deferred<T>** запускает блок кода асинхронно и позволяет вернуть результат
 - Можно подождать результата с помощью метода Deferred.await()
 - Метод await() – это тоже suspend-метод
- **withContext(): T** запускает блок кода и **ожидает** результат этого блока

Custom scope

```
class DataRepository(appScope: CoroutineScope) {  
  
    private val repoScope =  
        appScope.childScope(context = Dispatchers.Default)  
    }  
  
    fun CoroutineScope.childScope(  
        context: CoroutineContext = EmptyCoroutineContext,  
    ): CoroutineScope {  
        // получаем Job из родительского CoroutineScope  
        val parentJob = checkNotNull(coroutineContext[Job])  
        return CoroutineScope(  
            coroutineContext + Job(parent = parentJob) + context  
        )  
    }  
}
```

LivecycleScope — скоуп связанный и жтзненным циклом фрагмента или активити

этот скоуп это поле класса Lifecycle типа LifecycleCoroutineScopeImpl который является обычным корутин скоупом за исключением того что у него есть функция onStateChanged которое проверяет состояние LifecycleOwner и если он **DESTROYED** то отменяет корутин скоуп

ViewModelScope - скоуп связанный и жтзненным циклом вьюмодели

этот скоуп мы получаем из объекта viewModel по тегу из **HashMap<String, Object>** , объект является классом CloseableCoroutineScope с функцией **close** в которой мы и отменяем корутину при завершении работы viewModel

GlobalScope — срок жизни приложения

runBlocking{} - блокирует поток по этому мы его не используем

- **Dispatchers.Main** - Use this dispatcher to run a coroutine on the main Android thread. This should be used only for interacting with the UI and performing quick work. Examples include calling `suspend` functions, running Android UI framework operations, and updating `LiveData` objects.
- **Dispatchers.IO** - This dispatcher is optimized to perform disk or network I/O outside of the main thread. Examples include using the [Room component](#), reading from or writing to files, and running any network operations.

Dispatcher Default — пул фоновых потоков без большого количества операций ввода вывода — определен по дефолту)

Unconfimed Dispatcher - не привязан к потоку и будет выполнено в том потоке его создано и запущено

Корутина в вычислении не может быстро закончиться так что для её завершения мы либо используем внутри блок try — catch что бы ловить ошибку CancellationException, либо мы можем внутри корутин проверять активна ли она (is Active)

```
@Test
fun standardTest() = runTest {
    val userRepo = UserRepository()

    launch { userRepo.register("Alice") }
    launch { userRepo.register("Bob") }
    advanceUntilIdle() // Yields to perform the registrations

    assertEquals(listOf("Alice", "Bob"), userRepo.getAllUsers()) // ✅ Passes
}
```

```
class RepositoryTestWithRule {
    @get:Rule
    val mainDispatcherRule = MainDispatcherRule()

    private val repository = Repository(mainDispatcherRule.testDispatcher)

    @Test
    fun someRepositoryTest() = runTest { // Takes scheduler from Main
        // Any TestDispatcher created here also takes the scheduler from Main
        val newTestDispatcher = StandardTestDispatcher()

        // Test the repository...
    }
}
```

Параллельное выполнение корутин (а и б тогда выполняться параллельно)

```
coroutineScope { this: CoroutineScope
    launch(Dispatchers.IO) { a() }
    launch(Dispatchers.IO) { b() } ^cc
}
```

Flows are cold

Flows are *cold* streams similar to sequences — the code inside a [flow](#) builder does not run until the flow is collected.

Общие потоки никогда не завершаются. Другими словами, когда вы вызываете `Flow.collect()` в общем потоке, вы не собираете все его события. Вместо этого вы подписываетесь на события, которые генерируются, пока существует эта подписка.

Хотя это и означает, что обращение к `Flow.collect()` в общих потоках не завершаются нормально, подписку все же можно отменить. Как и следовало ожидать, эта отмена происходит путем отмены сопрограммы.

Заметка

Операторы усечения потока, такие как `Flow.take(count: Int)`, могут принудительно завершить общий поток.

Cold Flows

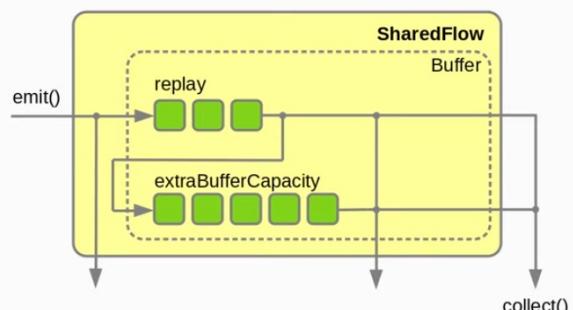
- Находятся в “спящем”, неактивном состоянии
- Активируются лениво только после вызова “терминального” оператора
- Создаются с помощью Flow Builders:
 - `flowOf(...)` / `.asFlow()`
 - `flow {}`
 - `callbackFlow {}` / `channelFlow {}`

Hot Flows

- Находятся всегда в активном состоянии
- Им всё-равно, слушает их кто-то или нет
- Создаются с помощью операторов или методов-конструкторов:
 - `.shareIn` / `MutableSharedFlow()`
 - `.stateIn` / `MutableStateFlow()`

SharedFlow

- Бесконечный Hot Flow
- Создается двумя способами:
 - `MutableSharedFlow(replay: Int, extraBufferCapacity: Int, onBufferOverflow = SUSPEND | DROP_LATEST | DROP_OLDEST)`
 - Комбинацией операторов `Flow<T>.buffer(...).shareIn(...)`
 - `buffer(bufferCapacity, onBufferOverflow)` - optionalный оператор
 - `shareIn(scope: CoroutineScope, started = Eagerly | Lazily | WhileSubscribed(stopTimeoutMillis, replayExpirationMillis), replay: Int = 0)`



Метододы :

Collect {value → }	Аналог subscribe получает значения
flowOn(Dispatcher)	При меняется только к цепочке сверху и изменяет контекст в котором происходит процесс (Dispatcher)
emit(value)	Передаёт значение в поток
Catch { e -> emit("Caught \$e") }	Аналог onError ()
onEach{value ->}	Аналог OnNext {}
flatMapConcat	Объединяет фловы так что бы они шли один за другим
Zip	Объединяют соответствующие значения потоков (одно значение сопоставляет другому и выдаёт мапу)
Combine	т соответствующие значения потоков не дожидаясь новой пары а сопоставляя каждому значение нового https://kotlinlang.org/docs/flow.html#combine

Channels

Есть два вида билдеров для двух видов каналов :

1) Канал ресивер

```
val channel: ReceiveChannel<Int> =  
    scope.produce {  
        var x = 1  
        while (true) {  
            send(x++)  
        }  
    }  
  
    scope.launch {  
        val data = channel.receive()  
    }
```

2) Канал актор :

```
    val actor: SendChannel<Any> = scope.actor {
        -> val data = receive()
            writeStringToFile(data)
    }

    scope.launch {
        -> actor.send(data)
    }
```

Flow vs channel

Flow являются обычными холодными или горячими потоками, они используются для внедрения реактивности и работу с потоками , **channel** используются для обеспечения безопасной работы с несколькими корутинами и обмена данными между ними

Крастно чёрное дерево

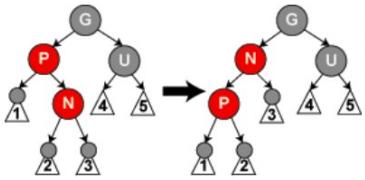
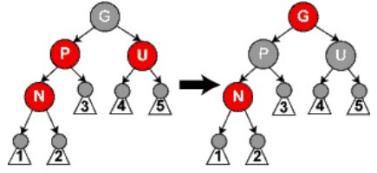
1. Узел может быть либо красным, либо чёрным и имеет двух потомков;
2. Корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;
3. Все листья, не содержащие данных — чёрные.
4. Оба потомка каждого красного узла — чёрные.
5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

Вставка

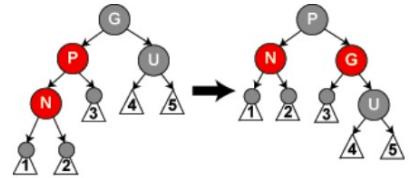
Случай 1: Текущий узел **N** в корне дерева. В этом случае, он перекрашивается в чёрный цвет, чтобы оставить верным Свойство 2 (Корень — чёрный). Так как это действие добавляет один чёрный узел в каждый путь, Свойство 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) не нарушается.

Случай 2: Предок **P** текущего узла чёрный, то есть Свойство 4 (Оба потомка каждого красного узла — чёрные) не нарушается. В этом случае дерево остаётся корректным. Свойство 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) не нарушается, потому что текущий узел **N** имеет двух чёрных листовых потомков, но так как **N** является красным, путь до каждого из этих потомков содержит такое же число чёрных узлов, что и путь до чёрного листа, который был заменен текущим узлом, так что свойство остается верным.

Случай 3: Если и родитель **P**, и дядя **U** — красные, то они оба могут быть перекрашены в чёрный, и дедушка **G** станет красным (для сохранения свойства 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов)). Теперь у текущего красного узла **N** чёрный родитель. Так как любой путь через родителя или дядю должен проходить через дедушку, число чёрных узлов в этих путях не изменится. Однако, дедушка **G** теперь может нарушить свойства 2 (Корень — чёрный) или 4 (Оба потомка каждого красного узла — чёрные) (свойство 4 может быть нарушено, так как родитель **G** может быть красным). Чтобы это исправить, вся процедура рекурсивно выполняется на **G** из случая 1.



Случай 4: Родитель **P** является красным, но дядя **U** — чёрный. Также, текущий узел **N** — правый потомок **P**, а **P** в свою очередь — левый потомок своего предка **G**. В этом случае может быть произведен поворот дерева, который меняет роли текущего узла **N** и его предка **P**. Тогда, для бывшего родительского узла **P** в обновленной структуре используем случай 5, потому что Свойство 4 (Оба потомка любого красного узла — чёрные) все ещё нарушено. Вращение приводит к тому, что некоторые пути (в поддереве, обозначенном «1» на схеме) проходят через узел **N**, чего не было до этого. Это также приводит к тому, что некоторые пути (в поддереве, обозначенном «3») не проходят через узел **P**. Однако, оба эти узла являются красными, так что Свойство 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) не нарушается при вращении. Однако Свойство 4 всё ещё нарушается, но теперь задача сводится к Случаю 5.



Случай 5: Родитель **P** является красным, но дядя **U** — чёрный, текущий узел **N** — левый потомок **P** и **P** — левый потомок **G**. В этом случае выполняется поворот дерева на **G**. В результате получается дерево, в котором бывший родитель **P** теперь является родителем и текущего узла **N** и бывшего дедушки **G**. Известно, что **G** — чёрный, так как его бывший потомок **P** не мог бы в противном случае быть красным (без нарушения Свойства 4). Тогда цвета **P** и **G** меняются и в результате дерева удовлетворяет Свойству 4 (Оба потомка любого красного узла — чёрные). Свойство 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) также остается верным, так как все пути, которые проходят через любой из этих трех узлов, ранее проходили через **G**, поэтому теперь они все проходят через **P**. В каждом случае, из этих трёх узлов только один окрашен в чёрный.