План занятия

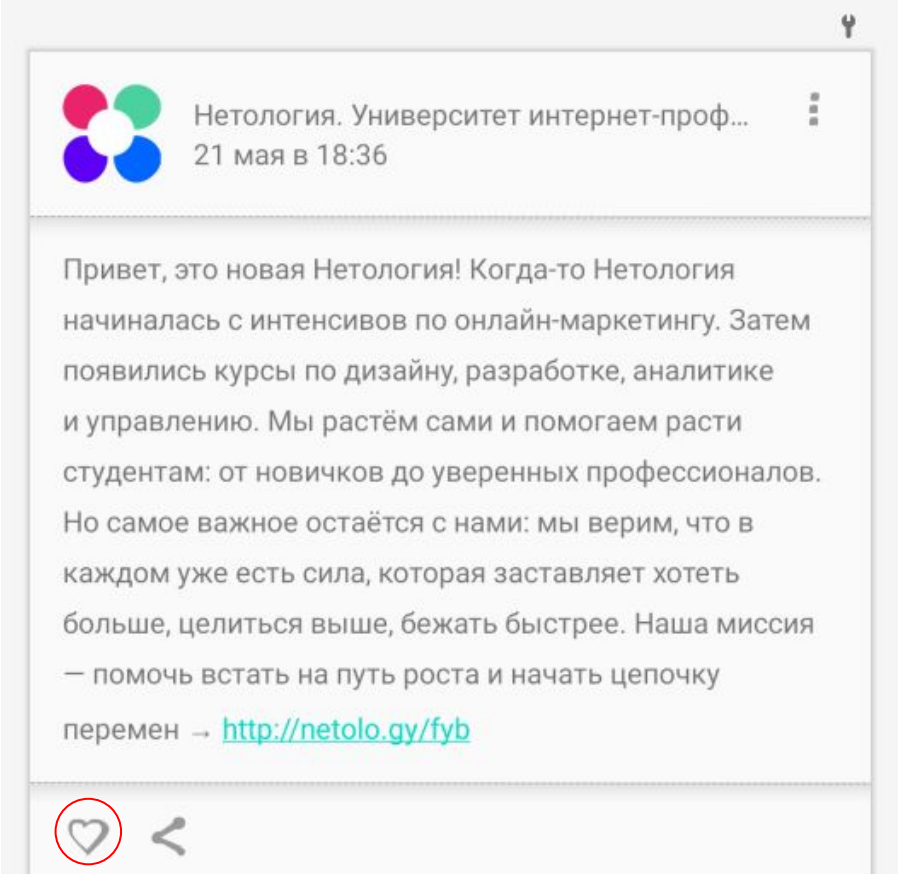
1. Проблема
2. MVVM
3. Реализация
4. Итоги

ПРОБЛЕМА

ПРОБЛЕМА

На прошлой лекции мы реализовали небольшое приложение.

Несмотря на то, что приложение небольшое, уже вырисовывается основной вопрос: насколько правильно то, что весь наш код сосредоточен в функции onCreate?



ПРОБЛЕМА

Мы получили классическую проблему всё в одном:

- данные
- жизненный цикл
- бизнес-логика

разнородные задачи

(логика обработки данных — при клике на лайк меняем флаг liked)

Если сосредоточить много разнородных задач в одном классе, то этот класс:

- станет «большим» и сложным для поддержки,
- будет плохо поддаваться тестированию.

ВОПРОС

? *Как правильно организовать распределение задач?*

АРХИТЕКТУРЫ

Для ответа на вопрос о «правильной» организации распределения задач между различными сущностями приложения придумывают различные архитектуры (проще говоря, подходы).

Например, в Android для этого существуют:

- MVP (Model View Presenter)
- MVI (Model View Intent)
- MVVM (Model View ViewModel)

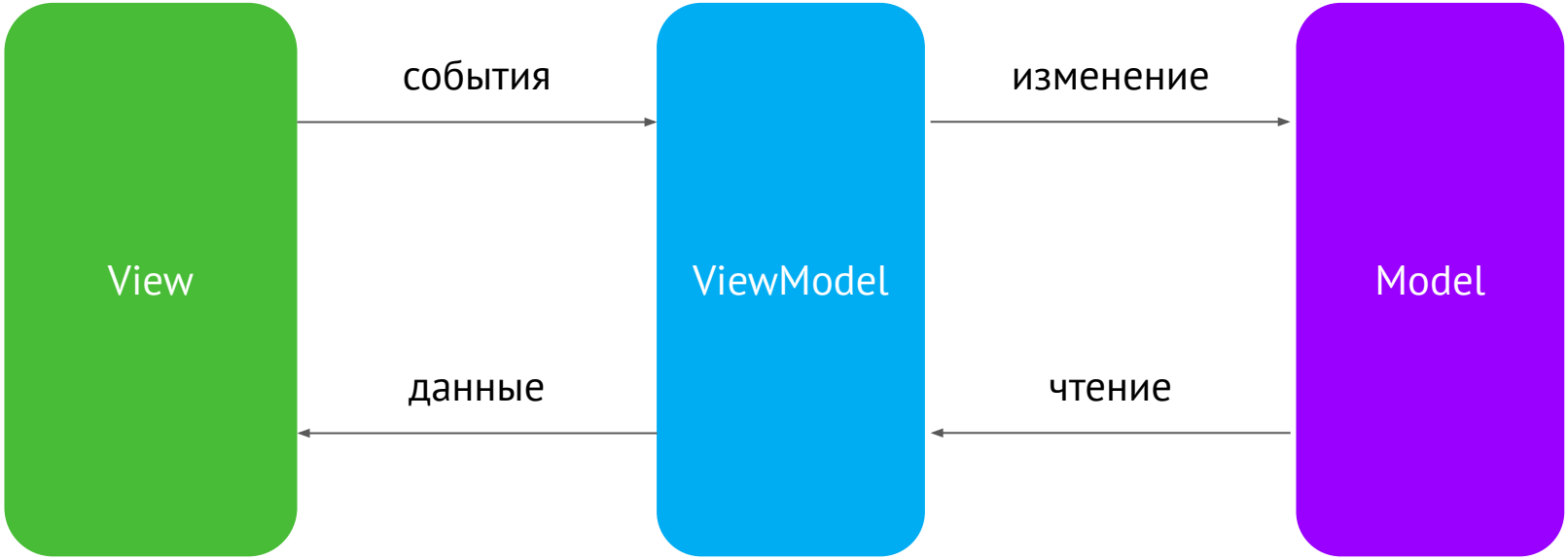
АРХИТЕКТУРЫ

Ключевая идея: мы разделяем всё приложение на слои, каждый из которых отвечает за одну задачу: или отображение данных, или бизнес-логику, или что-то другое.

Поскольку Google активно продвигает MVVM, то и использовать в рамках курса мы будем именно её.

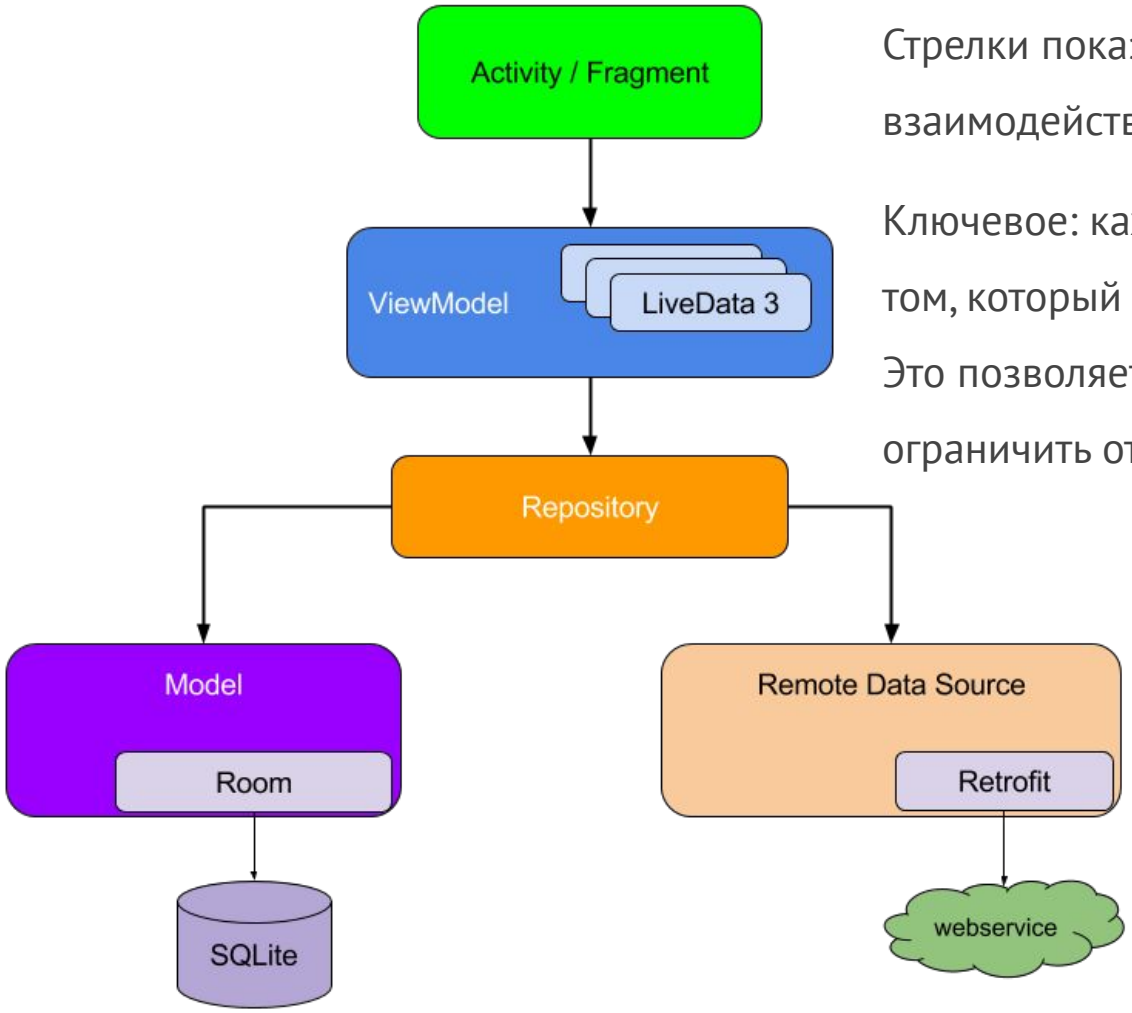
MVVM

MVVM



MVVM

направление взаимодействия



Стрелки показывают направление взаимодействия.

Ключевое: каждый слой знает только о том, который непосредственно под ним. Это позволяет менять реализации и ограничить ответственность.

Источник: developer.android.com

VIEWMODEL

С Activity мы уже немного знакомы (с Fragment разберёмся чуть позже), поэтому смотрим на ViewModel.

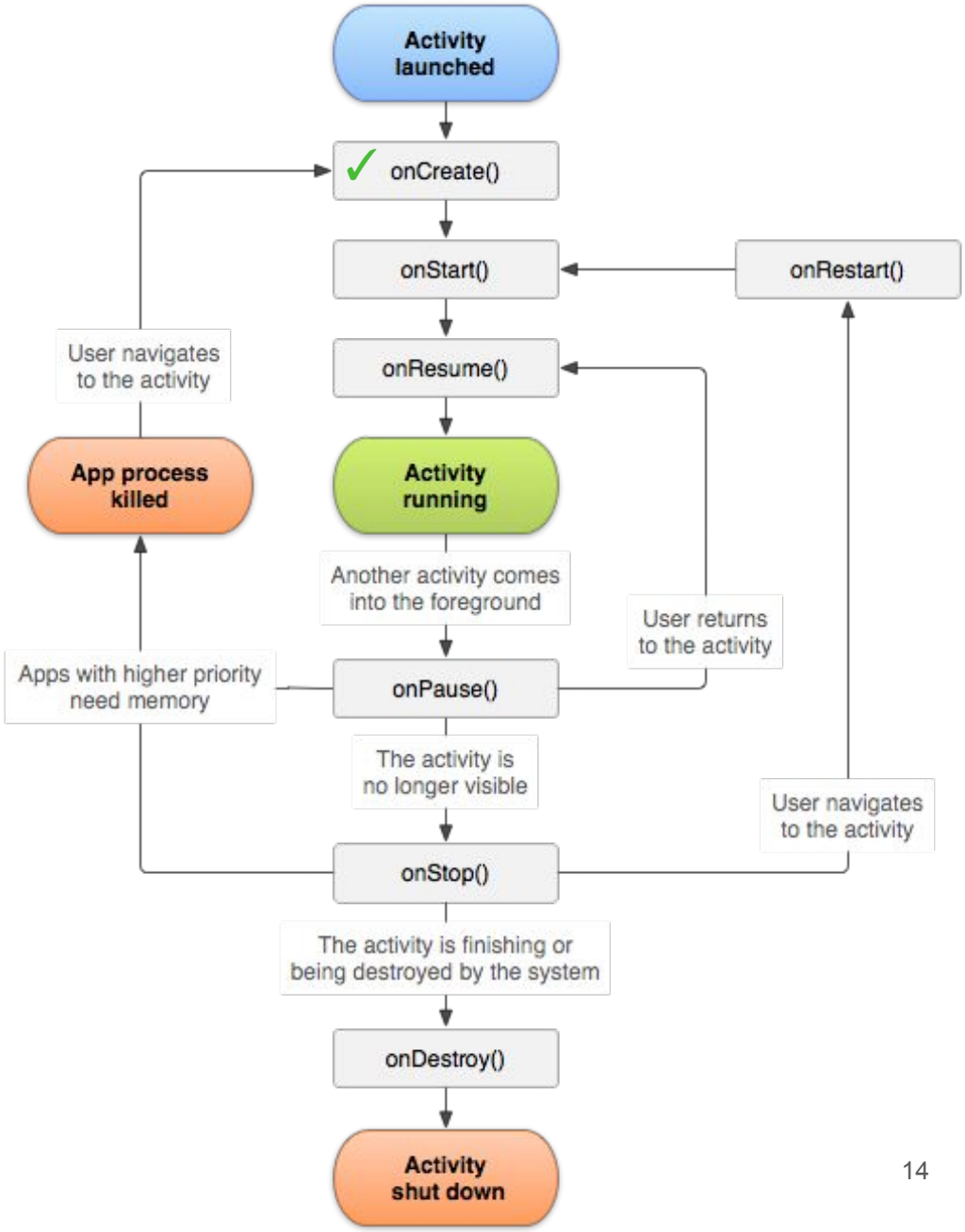
ViewModel — это класс, предназначенный для хранения и управления данными для UI в соответствии с жизненным циклом.

LIFECYCLE

Lifecycle (жизненный цикл) — это серия фаз, через которые проходят компоненты Android (по факту Android вызывает переопределённые функции).

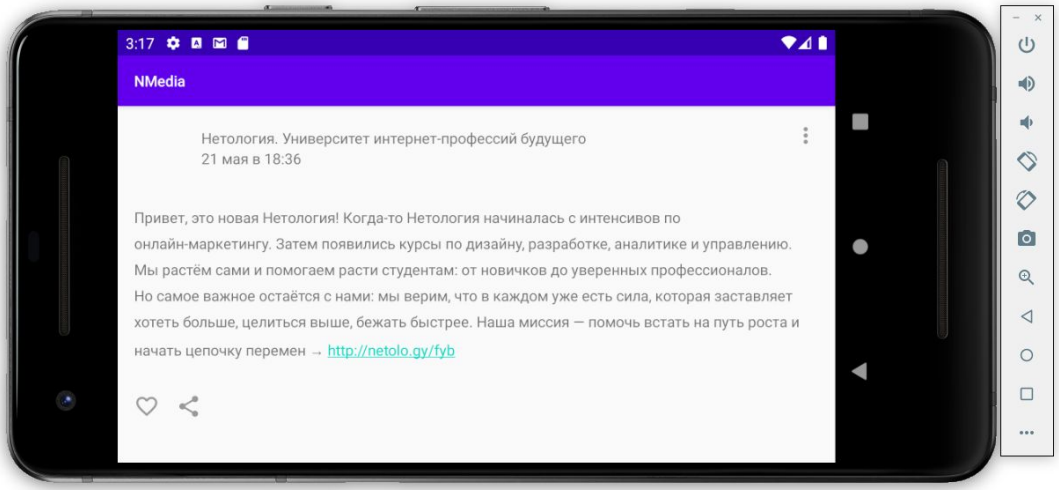
Например, сбоку представлен жизненный цикл Activity:

Т.е. в некоторых ситуациях происходит уничтожение объекта.



CONFIGURATION CHANGES

Если мы возьмём наше приложение с лайками, поставим лайк и перевернём устройство, то всё вернётся в изначальное состояние (как будто мы и не ставили лайк):



Изменение ориентации — один из примеров Configuration Change, которое приводит к тому, что Activity (не всё приложение) «убивается» и пересоздаётся заново (поведение по умолчанию).

LIFECYCLE & CONFIGURATION CHANGES

Про Lifecycle и Configuration Changes мы будем говорить отдельно.

Пока важно, что мы хотим сохранять данные, а не «терять» их при каждом пересоздании Activity.

РЕАЛИЗАЦИЯ

ЗАВИСИМОСТИ

Нам понадобятся следующие зависимости* (актуальные версии можно посмотреть здесь):

```
def arch_version = "2.1.0"
def lifecycle_version = "2.3.0"
```

объявление переменной в Groovy

```
// ViewModel
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
// LiveData
implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"
// optional - Test helpers for LiveData
testImplementation "androidx.arch.core:core-testing:$arch_version"
```

строка с подстановкой значения
(кавычки обязательно должны быть двойные)

Примечание*: см. в репозитории с кодом полную версию.

JAVA 8

Кроме того, для использования новых возможностей, понадобится включить поддержку Java 8:

```
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}
kotlinOptions {
    jvmTarget = "1.8"
}
```

REPOSITORY

Первое, с чего мы начнём — это Repository.

Repository — это шаблон проектирования, позволяющий взаимодействовать с чем-либо как с коллекцией объектов.

В сегодняшней лекции у нас только один объект (на следующей их станет несколько), поэтому мы позволим себе отойти от традиционного понимания и реализуем репозиторий для одного объекта (см. следующий слайд).

POST

```
data class Post(
    val id: Long,
    val author: String,
    val content: String,
    val published: String,
    val likedByMe: Boolean,
)
```

избавились от var

REPOSITORY

```
interface PostRepository {
    fun get(): LiveData<Post>
    fun like()
}
```

интерфейс

```
class PostRepositoryInMemoryImpl : PostRepository {
    private var post = Post(
        id = 1,
        author = "Нетология. Университет интернет-профессий будущего",
        content = "Привет, это новая Нетология! Когда-то Нетология начиналась
        published = "21 мая в 18:36",
        likedByMe = false
    )
    private val data = MutableLiveData(post)

    override fun get(): LiveData<Post> = data
    override fun like() {
        post = post.copy(likedByMe = !post.likedByMe)
        data.value = post
    }
}
```

конкретная реализация

(in memory)

setter у MutableData

setValue

```
public void setValue (T value)
```

Sets the value. If there are active observers, the value will be dispatched to them.

LIVEDATA & MUTABLELIVEDATA

LiveData — абстрактный класс, который позволяет различным компонентам Android «подписываться» на обновления и получать эти уведомления только тогда, когда это имеет для них смысл (например, когда они отображаются на экране смартфона).

MutableLiveData — одна из реализаций LiveData.

Ключевая идея: кто-то должен подписаться на LiveData, чтобы получать уведомления об изменении (когда мы сделали setValue, т.е. вызвали setter, который в Kotlin выглядит как установка значения property).

VIEWMODEL

```
class PostViewModel : ViewModel() {
    // упрощённый вариант
    private val repository: PostRepository = PostRepositoryInMemoryImpl()
    val data = repository.get()
    fun like() = repository.like()
}
```

ViewModel отвечает за хранение данных, относящихся к UI, и за связывание UI с бизнес-логикой. В нашем случае, вся бизнес-логика хранится в Repository, именно он отвечает за логику работы like.

Обратите внимание, ViewModel хранит data (в виде LiveData — не изменяемо), потому что у репозитория могут быть разные реализации, а не только in memory.

VIEWMODEL

Q: Зачем вообще нужен репозиторий?

A: В реальных приложениях данные редко хранятся в памяти, чаще всего они хранятся либо в постоянном хранилище (файлах, базах данных), либо на удалённых ресурсах (веб-сервис и т.д.). Именно за это и отвечает репозиторий (знание о том, где и как реально хранятся данные). На данный момент он может показаться лишним, но уже через пару лекций мы увидим, что если не выделить слои сразу, потом придётся переписывать достаточно много кода.

VIEWMODEL

Q: Почему ViewModel просто не может хранить MutableLiveData?

A: Потому что эти данные реально принадлежат репозиторию и только он отвечает за их изменение. Если бы это были какие-то другие данные, которые не требовалось бы хранить, например, поисковый запрос (мы ищем что-то среди постов) или выбранный в данный момент пост из списка постов, то такие данные вполне можно хранить в самой ViewModel без привлечения репозитория. Всё зависит от бизнес-логики (например, в приложениях вроде GMail и Telegram, даже неотправленные сообщения сохраняются на сервере).

ACTIVITY

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        val viewModel: PostViewModel by viewModels()
        viewModel.data.observe( owner: this) { post ->
            with(binding) {  this: ActivityMainBinding
                author.text = post.author
                published.text = post.published
                content.text = post.content
                like.setImageResource(
                    if (post.likedByMe) R.drawable.ic_liked_24 else R.drawable.ic_like_24
                )
            }
        }
        binding.like.setOnClickListener {  it: View!
            viewModel.like()
        }
    }
}
```

подписка на обновление

DELEGATION

```
val viewModel: PostViewModel by viewModels()
```

С такой конструкцией мы ещё не встречались.

by в Kotlin означает делегирование*. Т.е. мы предоставляем кому-то другому (объекту или функции) выполнить определённую работу.

В данном случае — инициализировать model. По факту, происходит следующее: у объекта, возвращаемого функцией viewModels, вызывается функция getValue (или property value).

Примечание*: делегировать можно не только инициализацию переменных и свойств, но и реализацию функций (для наследования и реализации интерфейсов).

DELEGATION

```
/**
 * Returns a [Lazy] delegate to access the ComponentActivity's ViewModel, if [factoryProducer]
 * is specified then [ViewModelProvider.Factory] returned by it will be used
 * to create [ViewModel] first time.
 *
 * ```
 * class MyComponentActivity : ComponentActivity() {
 *     val viewmodel: MyViewModel by viewmodels()
 * }
 * ```
 *
 * This property can be accessed only after the Activity is attached to the Application,
 * and access prior to that will result in IllegalArgumentException.
 */
@MainThread
inline fun <reified VM : ViewModel> ComponentActivity.viewModels(
    noinline factoryProducer: (() -> Factory)? = null
): Lazy<VM> {...}
```

DELEGATION

Итак, давайте обсуждать:

- @MainThread — у нас появляются потоки (мы кратко о них упоминали, когда проходили обработку событий), мы к ним вернёмся, когда будем проходить загрузку данных по сети.
- inline/noinline и reified — указание компилятору на необходимость "встроить" вызов функции (т.е. убрать вызов и превратить его в обычный код — inline), при этом мы получаем возможность сохранить информацию о типе (reified).
- lazy — ленивая инициализация (инициализация происходит не в момент объявления, а в момент первого вызова).

DELEGATION

```
class ViewModelLazy<VM : ViewModel> (
    private val viewModelClass: KClass<VM>,
    private val storeProducer: () -> ViewModelStore,
    private val factoryProducer: () -> ViewModelProvider.Factory
) : Lazy<VM> {
    private var cached: VM? = null

    override val value: VM
        get() {
            val viewModel = cached                    ● breakpoint
            return if (viewModel == null) {
                val factory = factoryProducer()
                val store = storeProducer()
                ViewModelProvider(store, factory).get(viewModelClass.java).also { it: VM
                    cached = it
                }
            } else {
                viewModel
            }
        }

    override fun isInitialized() = cached != null
}
```

DELEGATION

Если мы запустимся под отладчиком, установив точку остановки как на предыдущем слайде, то:

здесь не инициализировалась наша модель (инициализируется лениво на следующей строке)

```
val viewModel: PostViewModel by viewModels()
viewModel.data.observe( owner: this) { post ->
```

DELEGATION

Идём дальше:

чтобы получать уведомления только тогда, когда Activity находится в «активном» состоянии (Lifecycle.State.STARTED или Lifecycle.State.RESUMED)

ИТОГИ

ИТОГИ

Сегодня мы обсудили вопросы организации кода (как хранить данные в приложении). Достаточно долго разработчиками Android не предлагалось никакого подхода и инструментов для следования ему, в связи с чем и возникли различные варианты.

На сегодняшний момент рекомендуемым является MVVM*, поэтому мы будем придерживаться его.

Примечание*: это не значит, что это единственно верный и подходящий во всех случаях подход. Но он будет самым распространённым.