



План занятия

1. [Задача](#)
2. [Удаление](#)
3. [Добавление](#)
4. [Редактирование](#)
5. [Итоги](#)



ЗАДАЧА

ЗАДАЧА

Мы научились отображать коллекцию элементов, самое время научиться делать CRUD-операции:

- создание постов
- получение постов (уже сделали)
- обновление (сделали частично - лайк)
- удаление



УДАЛЕНИЕ

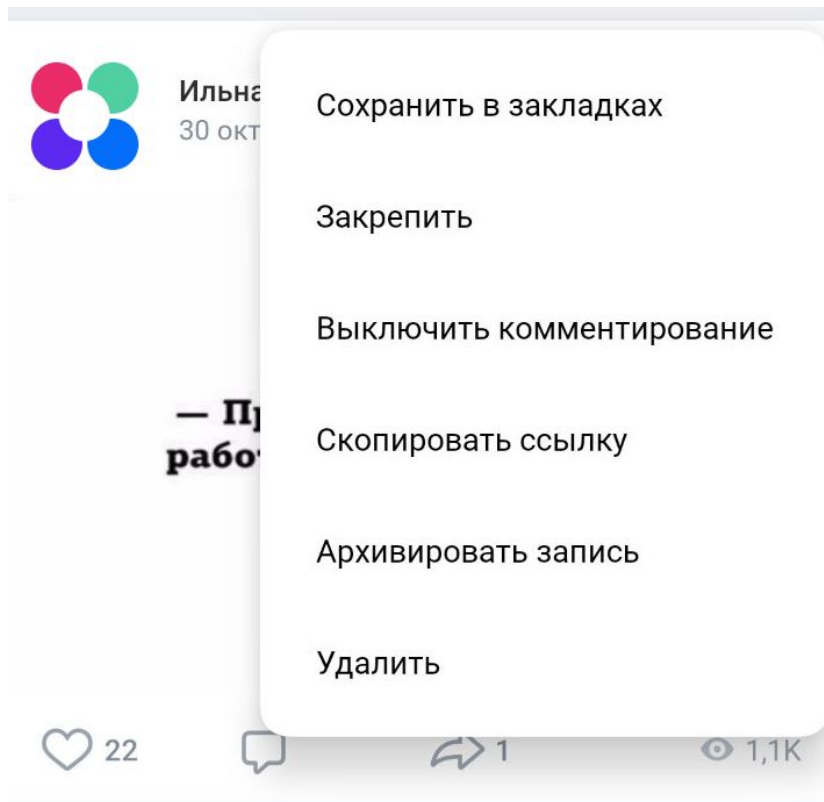
УДАЛЕНИЕ

Начнём с самого простого — удаление. Для этого достаточно сделать несколько вещей:

1. «Нарисовать» компонент, отвечающий за удаление.
2. Реализовать соответствующие коллбеки в [Adapter](#)'е.
3. Реализовать соответствующие методы во [ViewModel](#) и [Repository](#).

МЕНЮ

Удаление, как и некоторые другие не очень часто выполняемые действия, обычно прячут во всплывающее меню:



VIEWMODEL & REPOSITORY

```
class PostViewModel : ViewModel() {  
    // упрощённый вариант  
    private val repository: PostRepository = PostRepositoryInMemoryImpl()  
    val data = repository.getAll()  
    fun likeById(id: Long) = repository.likeById(id)  
    fun removeById(id: Long) = repository.removeById(id)  
}
```

```
interface PostRepository {  
    fun getAll(): LiveData<List<Post>>  
    fun likeById(id: Long)  
    fun removeById(id: Long)  
}
```

REPOSITORY IMPLEMENTATION

```
class PostRepositoryInMemoryImpl : PostRepository {  
    private var posts = listOf(...)  
  
    private val data = MutableLiveData(posts)  
  
    override fun getAll(): LiveData<List<Post>> = data  
    override fun likeById(id: Long) {...}  
  
    override fun removeById(id: Long) {  
        posts = posts.filter { it.id != id}  
        data.value = posts  
    }  
}
```

ADAPTER

```
typealias OnLikeListener = (post: Post) -> Unit  
typealias OnRemoveListener = (post: Post) -> Unit
```

```
class PostsAdapter(  
    private val onLikeListener: OnLikeListener,  
    private val onRemoveListener: OnRemoveListener,  
    ) : ListAdapter<Post, PostViewHolder>(PostDiffCallback()) {...}
```



НАДВИГАЮЩИЕСЯ ПРОБЛЕМЫ

Пока подход с созданием `typealias`'ов и передачей `callback`'ов в конструктор выглядит вполне рабочим.

Но если посмотреть чуть дальше, то проблемы начнут проявляться при:

1. появлении нескольких адаптеров (они все в одном пакете),
2. увеличении количества `callback`'ов (что, если их нужно будет штук 10?).

Уже сейчас стоит об этом задуматься. Но для начала допишем весь код, т.к. нет смысла улучшать то, что не работает.

VIEWHOLDER

```
menu.setOnClickListener { it: View!  
    PopupMenu(it.context, it).apply { this: PopupMenu  
        inflate(R.menu.options_post)  
        setOnMenuItemClickListener { item ->  
            when (item.itemId) {  
                R.id.remove -> {  
                    onRemoveListener(post)  
                    true ^setOnMenuItemClickListener  
                }  
                else -> false ^setOnMenuItemClickListener  
            }  
        }  
    }.show()  
}
```

MENU

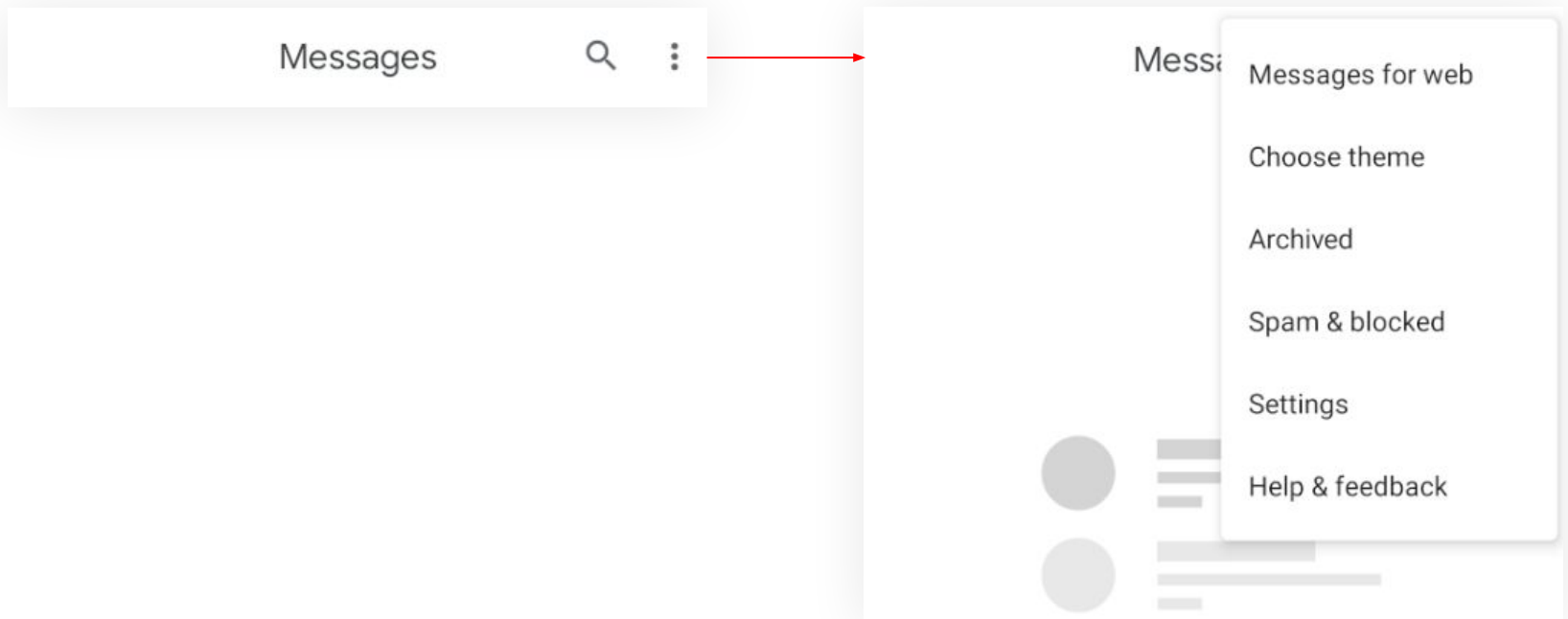
[PopupMenu](#) — специальный компонент, ответственный за показ всплывающего меню.

Вообще говоря, стоит различать [три вида "меню"](#), которые есть в Android:

- Options Menu
- Context Menu
- Popup Menu

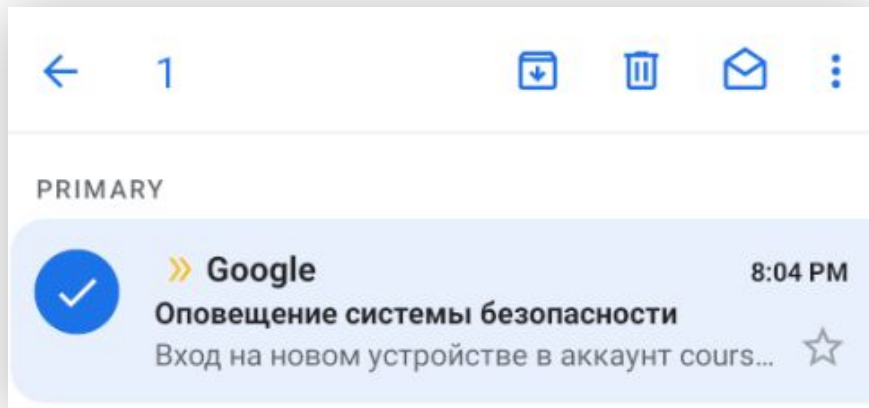
OPTIONS MENU

Показывается в верхней части приложения, часто содержит основные действия для Activity.



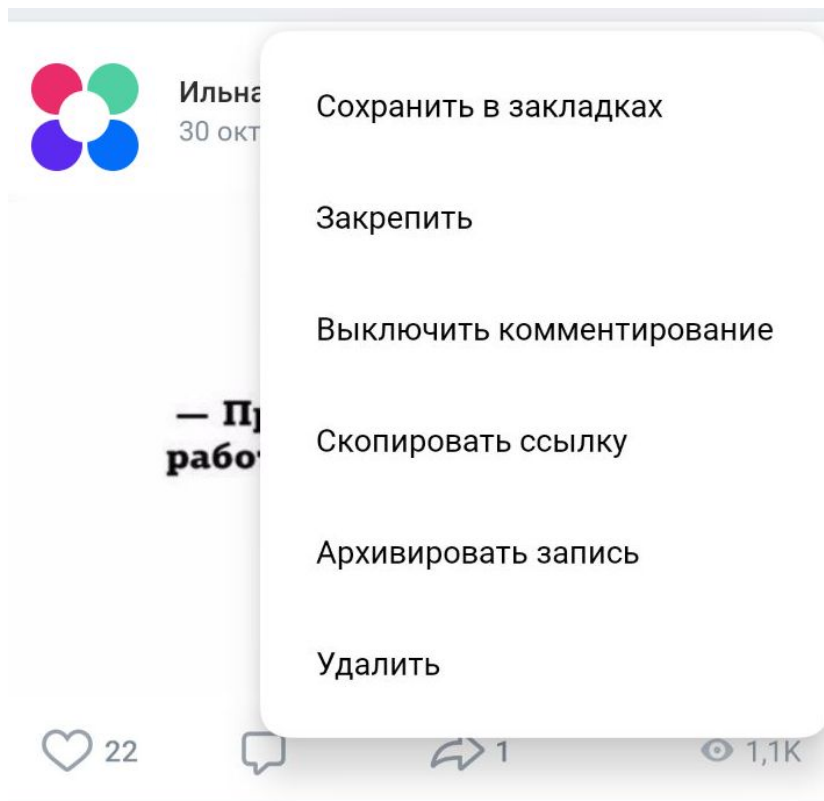
CONTEXT MENU

Отображается при выделении элемента либо в верхней части экрана (App Bar), либо рядом с самим элементом:



POPUP MENU

Отображается рядом с элементом, послужившим вызову этого меню:



POPUP MENU

Мы пошли по самому простому пути и реализовали всплывающее окно по клику на соответствующей кнопке:

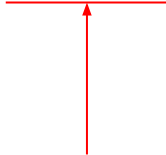
```
menu.setOnClickListener { it: View!
    PopupMenu(it.context, it).apply { this: PopupMenu
        inflate(R.menu.options_post) ← пункты меню (ресурс)
        setOnMenuItemClickListener { item ->
            when (item.itemId) {
                R.id.remove -> {
                    onRemoveListener(post)
                    true ^setOnMenuItemClickListener
                }
                else -> false ^setOnMenuItemClickListener
            }
        }
    }.show() ← показ меню
}
```

обработчик клика

MENU

XML-песчница:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/remove" android:title="Remove" />
</menu>
```



локализованная строка

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()  
        val adapter = PostsAdapter(  
            onLikeListener = { it: Post  
                viewModel.likeById(it.id)  
            },  
            onRemoveListener = { it: Post  
                viewModel.removeById(it.id)  
            }  
        )  
        binding.list.adapter = adapter  
        viewModel.data.observe(owner: this) { posts ->  
            adapter.submitList(posts)  
        }  
    }  
}
```

уже не очень удобно



ИТОГИ

Несмотря на все неудобства в коде, наше приложение вполне работает, а [RecyclerView](#) даже анимирует процесс удаления.



ДОБАВЛЕНИЕ

ЛЕНТА ПОСТОВ

С добавлением всё немного сложнее. Давайте сделаем следующим образом: текстовое поле внизу ленты, в которое пользователь может ввести текст и добавить пост (самое простейшее, что мы можем сделать, без всяких вложений и т.д.):



И разрешим мы редактировать только контент.

LAYOUT

В принципе, всё нам знакомо: Chaining из двух компонентов ([EditText](#) и [ImageButton](#)) и [Barrier](#) (top).

Но вот про сам [EditText](#) нужно поговорить подробнее.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".activity.MainActivity">

    <androidx.recyclerview.widget.RecyclerView...>

    <androidx.constraintlayout.widget.Barrier...>

    <EditText
        android:id="@+id/content"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:hint="@string/post_text"
        android:inputType="textMultiLine"
        android:background="@android:color/transparent"
        android:padding="@dimen/common_spacing"
        android:importantForAutofill="no"
        app:layout_constraintTop_toTopOf="@id/toolsTop"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toStartOf="@id/save"
        app:layout_constraintStart_toStartOf="parent"
    />

    <ImageButton...>

</androidx.constraintlayout.widget.ConstraintLayout>
```

EDITTEXT

[EditText](#) — это специальный компонент для ввода текста. С вводом текста в Android всегда всё было не очень (вы это увидите чуть позже).

Ключевые моменты:

- имеет подсказку для ввода (hint) — пропадает при получении фокуса;
- имеет тип ввода (inputType) — исходя из этого подстраивается клавиатура;
- может иметь [рекомендации к автодополнению](#) (мы отключили);
- по умолчанию имеет фон в виде нижней горизонтальной линии, который мы убираем.



ОБЩАЯ СХЕМА

Пойдём мы всё по той же схеме, что проходили на лекциях по Kotlin: создадим объект для хранения данных нового поста и будем заполнять его.

Заранее договоримся, что `id = 0` означает, что это новый пост.


```
private val empty = Post(
    id = 0,
    content = "",
    author = "",
    likedByMe = false,
    published = ""
)
```

data-объект для заполнения

```
class PostViewModel : ViewModel() {
    // упрощённый вариант
    private val repository: PostRepository = PostRepositoryInMemoryImpl()
    val data = repository.getAll()
    val edited = MutableLiveData(empty)
```

```
fun save() {
    edited.value?.let { it: Post
        repository.save(it)
    }
    edited.value = empty
}
```

функция сохранения

```
fun changeContent(content: String) {
    edited.value?.let { it: Post
        val text = content.trim()
        if (it.content == text) {
            return
        }
        edited.value = it.copy(content = text)
    }
}
```

функция изменения контента

```
fun likeById(id: Long) = repository.likeById(id)
fun removeById(id: Long) = repository.removeById(id)
```

```
}
```

REPOSITORY

```
interface PostRepository {  
    fun getAll(): LiveData<List<Post>>  
    fun likeById(id: Long)  
    fun save(post: Post)  
    fun removeById(id: Long)  
}
```



```
override fun save(post: Post) {  
    posts = listOf(  
        post.copy(  
            id = nextId++,  
            author = "Me",  
            likedByMe = false,  
            published = "now"  
        )  
    ) + posts  
    data.value = posts  
    return  
}
```

ACTIVITY

```
binding.save.setOnClickListener { it: View!  
    with(binding.content) { this: EditText  
        if (text.isNullOrBlank()) {  
            Toast.makeText(  
                context: this@MainActivity,  
                "Content can't be empty",  
                Toast.LENGTH_SHORT  
            ).show()  
            return@setOnClickListener  
        }  
  
        viewModel.changeContent(text.toString())  
        viewModel.save()  
  
        setText("")  
        clearFocus()  
        AndroidUtils.hideKeyboard( view: this)  
    }  
}
```

показ «всплывашки»



ФОКУС

Фокус ввода подразумевает под собой «выбор» какого-либо элемента для получения событий ввода.

В частности, когда поля ввода в фокусе, то в них устанавливается курсор и ввод с виртуальной клавиатуры попадает в них (поля ввода).

HIDE KEYBOARD

После сохранения поста необходимо скрывать клавиатуру, поэтому мы написали вспомогательную функцию, которая это делает:

```
object AndroidUtils {  
    fun hideKeyboard(view: View) {  
        val imm = view.context.getSystemService(Context.INPUT_METHOD_SERVICE) as InputMethodManager  
        imm.hideSoftInputFromWindow(view.windowToken, flags: 0)  
    }  
}
```

Отвечает за это специальный системный сервис. Чтобы его получить, нам нужен `context`. И соответствующий метод позволяет скрыть клавиатуру.

Особой логики тут нет, кроме того, что это одна из частей API, которая спроектирована «ужасно».



ДОБАВЛЕНИЕ



ДОБАВЛЕНИЕ

Добавление работает, но возникает вопрос: насколько корректно «валидировать» данные в Activity?

Пока мы оставим его без ответа, но вернёмся к нему, когда будем проходить интеграцию с backend'ом.



РЕДАКТИРОВАНИЕ



РЕДАКТИРОВАНИЕ

С редактированием всё сложнее: нужно во всплывающем меню сделать пункт «Изменить» и заполнить поле ввода данными поста.

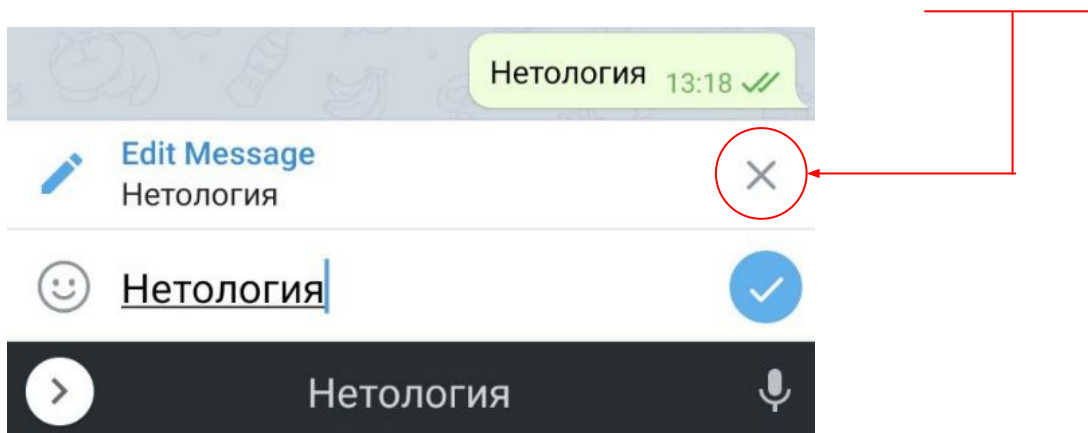
И здесь всплывает сразу несколько нюансов:

1. Что если пользователь уже ввёл что-то в поле ввода?
2. Что если пользователь захочет «отменить» редактирование
3. Все ли посты может редактировать пользователь (кстати, это же относилось к редактированию) — про это будем говорить отдельно.

НЮАНСЫ

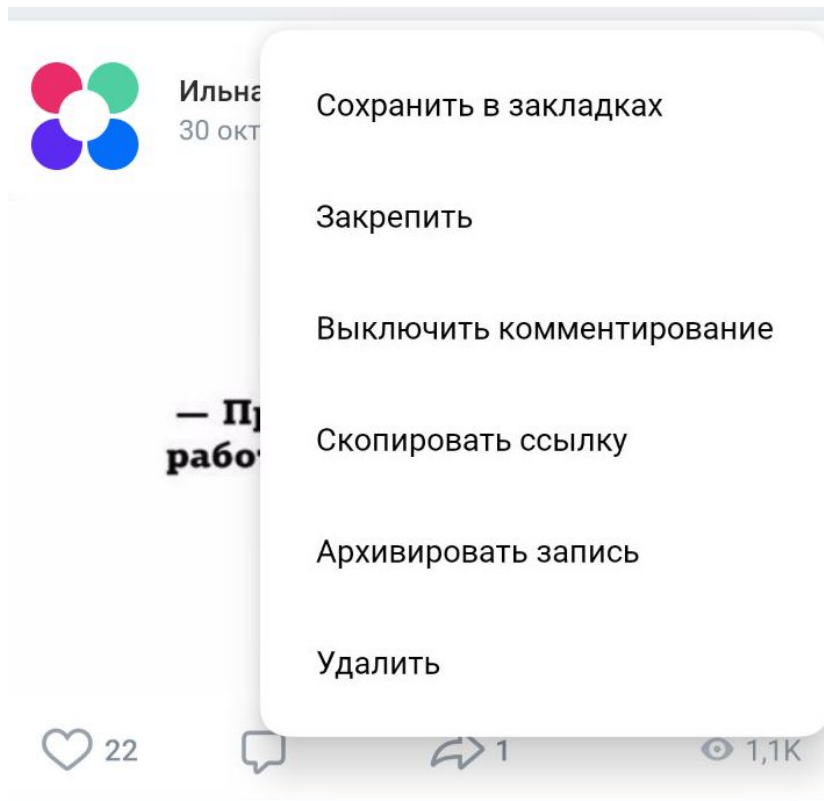
Самое правильное в таком случае — это посмотреть, как реализованы уже аналогичные приложения. Например, в Телеграм, если вы ввели что-то в поле ввода, а затем выбрали сообщение для редактирования, ваш первоначальный текст просто стирается (на момент написания лекции).

А для отмены редактирования появляется соответствующий управляющий элемент в виде крестика наверху:



ADAPTER

Во-первых, нужно добавить в [Adapter](#) callback — [onEditListener](#). И это будет уже немного перебором, поскольку если мы вспомним, то:



А в Telegram опций не меньше.

ADAPTER

Как один из вариантов — заведём специальный интерфейс с дефолтными реализациями функций:

```
interface OnInteractionListener {  
    fun onLike(post: Post) {}  
    fun onEdit(post: Post) {}  
    fun onRemove(post: Post) {}  
}
```

ADAPTER

Тогда мы сможем передавать нужные реализации:

```
val adapter = PostsAdapter(object : OnInteractionListener {  
    override fun onEdit(post: Post) {  
        viewModel.edit(post)  
    }  
  
    override fun onLike(post: Post) {  
        viewModel.likeById(post.id)  
    }  
  
    override fun onRemove(post: Post) {  
        viewModel.removeById(post.id)  
    }  
})
```

ADAPTER

А вызов будет выглядеть следующим образом:

```
menu.setOnClickListener { it: View!
    PopupMenu(it.context, it).apply { this: PopupMenu
        inflate(R.menu.options_post)
        setOnMenuItemClickListener { item ->
            when (item.itemId) {
                R.id.remove -> {
                    onInteractionListener.onRemove(post)
                    true ^setOnMenuItemClickListener
                }
                R.id.edit -> {
                    onInteractionListener.onEdit(post)
                    true ^setOnMenuItemClickListener
                }
                else -> false ^setOnMenuItemClickListener
            }
        }
    }.show()
}
```

```
class PostViewModel : ViewModel() {  
    // упрощённый вариант  
    private val repository: PostRepository = PostRepositoryInMemoryImpl()  
    val data = repository.getAll()  
    val edited = MutableLiveData(empty)  
  
    fun save() {  
        edited.value?.let { it: Post  
            repository.save(it)  
        }  
        edited.value = empty  
    }  
  
    fun edit(post: Post) {  
        edited.value = post  
    }  
  
    fun changeContent(content: String) {  
        val text = content.trim()  
        if (edited.value?.content == text) {  
            return  
        }  
        edited.value = edited.value?.copy(content = text)  
    }  
  
    fun likeById(id: Long) = repository.likeById(id)  
    fun removeById(id: Long) = repository.removeById(id)  
}
```

REPOSITORY

```
override fun save(post: Post) {  
    if (post.id == 0L) {  
        // TODO: remove hardcoded author & published  
        posts = listOf(post.copy(  
            id = nextId++,  
            author = "Me",  
            likedByMe = false,  
            published = "now"  
        )) + posts  
        data.value = posts  
        return  
    }  
  
    posts = posts.map { it: Post  
        if (it.id != post.id) it else it.copy(content = post.content)  
    }  
    data.value = posts  
}
```


ACTIVITY

```
viewModel.edited.observe( owner: this) { post ->
    if (post.id == 0L) {
        return@observe
    }
    with(binding.content) { this: EditText
        requestFocus()
        setText(post.content)
    }
}
```

как только `edited` меняется, обновляем текст



РЕДАКТИРОВАНИЕ



ОТМЕНА РЕДАКТИРОВАНИЯ

Возможность отмены редактирования мы оставим вам для самостоятельной реализации в рамках ДЗ.



ИТОГИ



ИТОГИ

Сегодня мы обсудили вопросы редактирования элементов в списке.

Теперь наша задача:

1. Стилизовать наше приложение
2. Научиться создавать разные экраны для пользователя