
План занятия

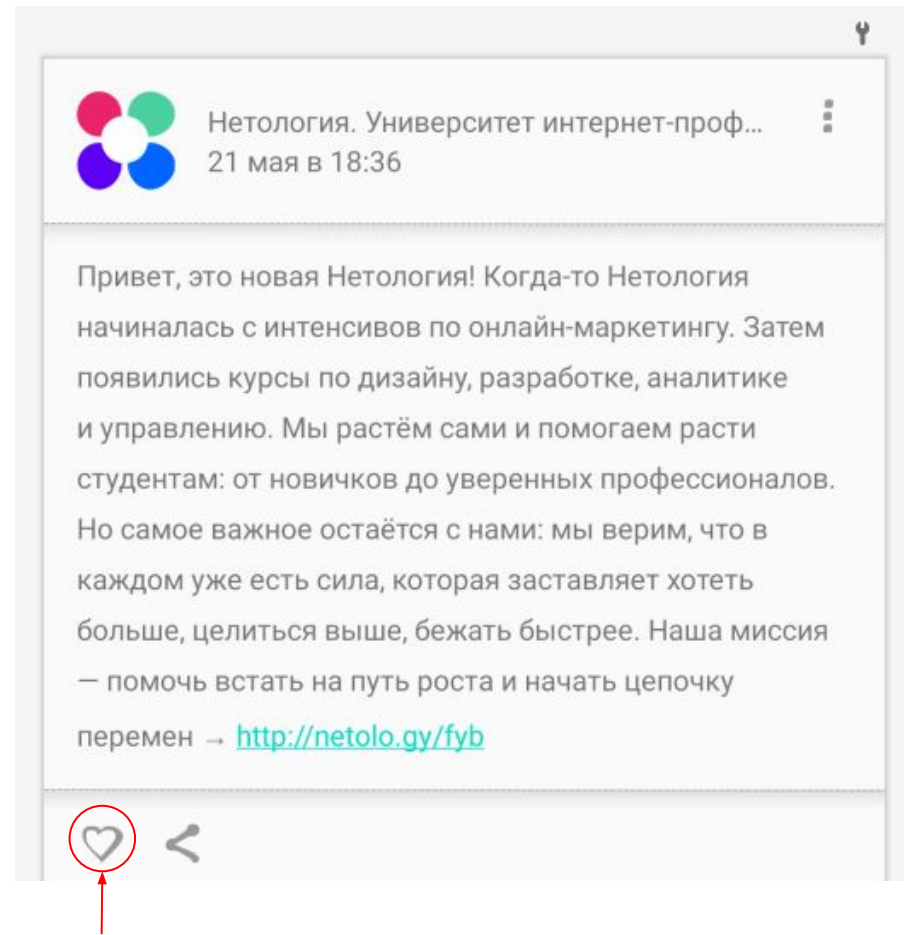
1. [Задача](#)
2. [UI](#)
3. [Listener](#)
4. [Kotlin Android Extensions](#)
5. [View Binding](#)
6. [Данные](#)
7. [Итоги](#)



ЗАДАЧА

ЗАДАЧА

Наша задача на сегодня:
научиться обрабатывать события
(клики) для поста нашей
социальной сети:



После клика должно стать красным

ЗАДАЧА

Для этого нам нужно ответить на следующие вопросы:

1. Как обрабатываются события?
2. Как менять свойства View (например, фоновое изображение)?

—
UI

UI

Event-Driven



Почти все UI-системы являются событийно-ориентированными: есть бесконечный цикл, который ожидает поступления событий, а после поступления их обрабатывает.

Событием может являться, например, клик, свайп и т.д.

Ключевое: мы не знаем, когда случится событие и случится ли оно вообще (это как с сообщениями в мессенджерах: вы не знаете, когда вам напишут, и напишут ли вообще).

UI

?

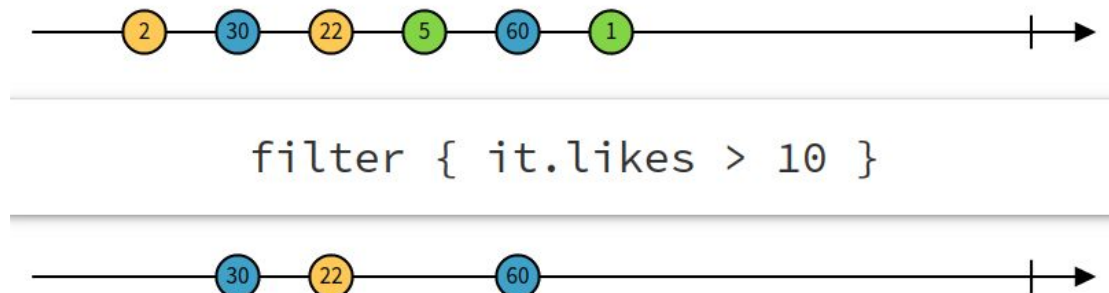
Как вы думаете, если мы не знаем, когда произойдёт событие и произойдёт ли вообще, как мы можем выполнить какой-то код в ответ на событие?

UI

Ключевая идея: мы можем подписаться на событие и нас оповестят тогда, когда это событие произойдёт (соответственно, если не произойдёт, то не оповестят).

Q: Что значит «подписаться» и «оповестят»?

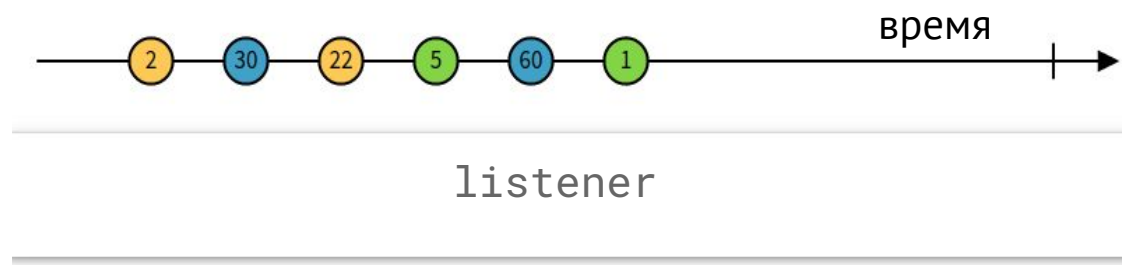
A: Вспомните Sequences в Kotlin — мы передавали lambda, которую запускали для каждого элемента:



изображения с rxmarbles.com

UI

А теперь представьте, что события — это те же самые объекты, просто распределённые во времени:



И мы можем передавать обработчик (в терминах Android его называют listener — слушатель), который будет запускаться системой при возникновении события.



LISTENER

LISTENER

Общий механизм устроен следующим образом: мы берём View, на котором хотим «слушать» события, и регистрируем свой listener:

- m 📄 `setOnClickListener(OnClickListener): void`
- m 📄 `setOnContextClickListener(OnContextClickListener): void`
- m 📄 `setOnCreateContextMenuListener(OnCreateContextMenuListener): void`
- m 📄 `setOnDragListener(OnDragListener): void`
- m 📄 `setOnFocusChangeListener(OnFocusChangeListener): void`
- m 📄 `setOnGenericMotionListener(OnGenericMotionListener): void`
- m 📄 `setOnHoverListener(OnHoverListener): void`
- m 📄 `setOnKeyListener(OnKeyListener): void`
- m 📄 `setOnLongClickListener(OnLongClickListener): void`
- m 📄 `setOnScrollChangeListener(OnScrollChangeListener): void`
- m 📄 `setOnSystemUiVisibilityChangeListener(OnSystemUiVisibilityChangeListener): void`
- m 📄 `setOnTouchListener(OnTouchListener): void`

методы View

LISTENER

Внутри всё устроено достаточно несложно:

```
/**
 * Register a callback to be invoked when this view is clicked. If this view is not
 * clickable, it becomes clickable.
 *
 * * @param l The callback that will run
 *
 * * @see #setClickable(boolean)
 */
public void setOnClickListener(@Nullable OnClickListener l) {
    if (!isClickable()) {
        setClickable(true);
    }
    getListenerInfo().mOnClickListener = l;
}
```

Фактически, есть приватное поле (оно во вложенном классе), которое и хранит listener.

LISTENER

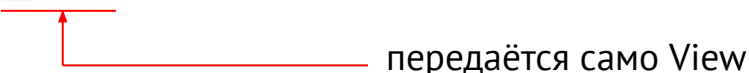
Сам OnClickListener представляет собой интерфейс с одним абстрактным методом (а значит может быть заменён функцией в Kotlin):

```
/**
 * Interface definition for a callback to be invoked when a view is clicked.
 */
public interface OnClickListener {
    /**
     * Called when a view has been clicked.
     *
     * @param v The view that was clicked.
     */
    void onClick(View v);
}
```

LISTENER

И сам вызов (будем пока считать, что вызывается Android):

```
public boolean performClick() {  
    // We still need to call this method to handle the cases where performClick() was called  
    // externally, instead of through performClickInternal()  
    notifyAutofillManagerOnClick();  
  
    final boolean result;  
    final ListenerInfo li = mListenerInfo;  
    if (li != null && li.mOnClickListener != null) {  
        playSoundEffect(SoundEffectConstants.CLICK);  
        li.mOnClickListener.onClick(v: this);  
        result = true;  
    } else {  
        result = false;  
    }  
  
    sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);  
  
    notifyEnterOrExitForAutoFillIfNeeded(true);  
  
    return result;  
}
```

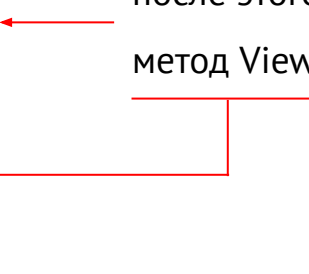


FINDVIEWBYID

Теперь, когда мы понимаем общую механику, можем установить listener на нашу кнопку. Но как до неё добраться?

Схема достаточно простая:

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main) ← после этого вызова можно использовать  
    }                                           метод View  
}  
  
@Nullable  
public final <T extends View> T findViewById( @IdRes @IdRes int id) {  
    if (id == NO_ID) {  
        return null;  
    }  
    return findViewTraversal(id);  
}
```



Q & A

Q: Зачем мы смотрим исходники, не проще ли просто почитать документацию?

A: Документация (JavaDoc'и) содержится и в исходниках, а кроме того, в исходниках вы можете прочесть то, чего нет в документации. И именно чтение исходников, а не документация или Google, помогут вам понять, **почему** в вашем приложении что-то работает не так, как вы хотите.

LISTENER

Собираем всё вместе:

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        findViewById<ImageButton>(R.id.like).setOnClickListener { it: View!  
            println("Clicked")  
        }  
    }  
}
```

?

Вопрос на засыпку: почему просто не сделать `onClickListener = ...?`

МЕНЯЕМ ИЗОБРАЖЕНИЕ

```
 class MainActivity : AppCompatActivity() {  
 override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
    findViewById<ImageButton>(R.id.like).setOnClickListener { it: View!  
        (it as ImageButton).setImageResource(R.drawable.ic_liked_24)  
    }  
}  
}
```

`it` — это `View`, а метод `setImageResource` есть только у `ImageButton` (на самом деле, у `ImageView`, а `ImageButton` лишь его наследник), поэтому мы используем `as`.

?

Как вы думаете, это хорошая идея — использовать `as`?

МЕНЯЕМ ИЗОБРАЖЕНИЕ

?

Как вы думаете, это хорошая идея — использовать `as`?

```
findViewById<ImageButton>(R.id.like).setOnClickListener { it: View!  
    if (it !is ImageButton) {  
        return@setOnClickListener  
    }  
  
    it.setImageResource(R.drawable.ic_liked_24)  
}
```

ЛОГИРОВАНИЕ

Здесь есть несколько подходов:


1. **Let it crash** — пусть приложение «упадёт» с Exception'ом, тогда мы получим об этом уведомление и поправим.
2. **Игнорирование** — тогда ничего не произойдёт, но, скорее всего, пользователь поставит низкую оценку и напишет, что кнопка не работает.
3. **Использовать сторонние библиотеки**, которые позволят exception/событие отловить, а нам — показать пользователю уведомление.

Пока мы будем использовать второй вариант, но уже на следующей лекции вернёмся к этому вопросу.

EXCEPTIONS

А теперь давайте немного поэкспериментируем:

поменяем на несовместимый тип
(TextView или R.id.author)



```
findViewById<ImageButton>(R.id.like).setOnClickListener { it: View!  
    if (it !is ImageButton) {  
        return@setOnClickListener  
    }  
  
    it.setImageResource(R.drawable.ic_liked_24)  
}
```

В обоих случаях мы «упадём» с ClassCastException. И IDE нас даже не предупредит, что мы ошиблись. Это плохо. Нужно искать более «безопасный» способ.



KOTLIN ANDROID EXTENSIONS

KOTLIN ANDROID EXTENSIONS

Kotlin Android Extensions (KAE) — это плагин для Gradle, который позволяет организовать типобезопасный доступ к компонентам из layout'a по их идентификаторам (именно поэтому мы использовали camelCase нотацию для имён идентификаторов).

Подключается в build.gradle (уже должен быть подключен):

```
apply plugin: 'com.android.application'  
apply plugin: 'kotlin-android'  
apply plugin: 'kotlin-android-extensions' ←
```

Важно: KAE в 2020 объявлено deprecated, но по-прежнему продолжает использоваться в существующих проектах.

SYNTHETIC

Далее в каждом **Activity** необходимо использовать специальный импорт вида:

```
import kotlinx.android.synthetic.main.activity_main.*;
```

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        like.setOnClickListener { it: View!  
            like.setImageResource(R.drawable.ic_liked_24)  
        }  
    }  
}
```

После чего нам станут доступны уже типизированные имена like, share и другие (по id, которые мы объявили в layout).

SYNTHETIC

С использованием KAE полностью пропадала необходимость использовать `findViewById`.

До недавнего времени это был самый популярный способ работы с View. И вы встретитесь с ним в большом количестве руководств и уже написанных приложений.

Но он обладает недостатком: некоторые идентификаторы могут присутствовать только в определённых конфигурациях (вспомните про `qualifiers` для ресурсов) и поэтому могут быть `null` или даже другого типа:

```
like.setOnClickListener { it: View!
```

Potential NullPointerException. The resource is missing in some of layout versions



VIEW BINDING

VIEW BINDING

Для устранения недостатков в КАЕ была разработана отдельная библиотека — [View Binding](#), которую мы и будем использовать на курсе.

В build.gradle необходимо прописать блок `buildFeatures` и `viewBinding`:

```
6  android {
7      compileSdkVersion 30
8
9      defaultConfig {...}
18
19      buildFeatures.viewBinding = true
20
21      buildTypes {...}
27      compileOptions {...}
31      kotlinOptions {jvmTarget = '1.8'}
34 }
```

← не забудьте после этого
синхронизировать проект



*BINDING

После включения нам будет предоставлен автоматически сгенерированный ***Binding** класс для каждого layout'a в нашем проекте (на самом деле — модуле app):

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

Т.е. у нас есть **activity_main.xml** и **activity_main.xml (land)** и для них сгенерировался **ActivityMainBinding**.

SETCONTENTVIEW

Метод `setContentView` перегруженный: до этого мы использовали `id` layout'a, сейчас же — `View`:

```
@Override  
public void setContentView(@LayoutRes int layoutResID) {...}
```

```
@Override  
public void setContentView(View view) { getDelegate().setContentView(view); }
```

```
@Override  
public void setContentView(View view, ViewGroup.LayoutParams params) {...}
```



LAYOUTINFLATER

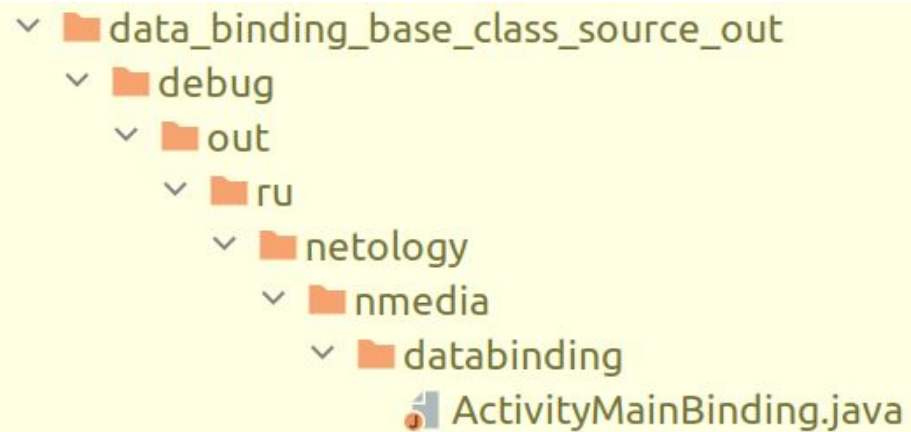
LayoutInflater — класс, умеющий из XML создавать иерархию View.

Существует в качестве property **Activity** (поскольку есть getter в соответствующем Java-классе).

Именно его использует сгенерированный класс для получения View.

*BINDING

Сам сгенерированный класс можно посмотреть в режиме Project:



```
public final class ActivityMainBinding implements ViewBinding {
```

```
    @NonNull
```

```
    private final ConstraintLayout rootView;
```

```
    @NonNull
```

```
    public final TextView author; ← не Nullable
```

```
    @NonNull
```

```
    public final ImageView avatar;
```

```
    ... // часть класса опущена
```

```
    /**
```

```
     * This binding is not available in all configurations.
```

```
     * <p>
```

```
     * Present:
```

```
     * <ul>
```

```
     *   <li>layout/</li>
```

```
     * </ul>
```

```
     *
```

```
     * Absent:
```

```
     * <ul>
```

```
     *   <li>layout-land/</li>
```

```
     * </ul>
```

```
     */
```

```
    @Nullable
```

```
    public final ImageButton like; ← Nullable, будем обязаны использовать ?. или !.
```

```
    ... // часть класса опущена
```


*BINDING

```
@NonNull
public static ActivityMainBinding bind(@NonNull View rootView) {
    // The body of this method is generated in a way you would not otherwise write.
    // This is done to optimize the compiled bytecode for size and performance.
    int id;
    missingId: {
        id = R.id.author;
        TextView author = rootView.findViewById(id);
        if (author == null) {
            break missingId;
        }

        ... // часть класса опущена

        id = R.id.like;
        ImageButton like = rootView.findViewById(id);

        ... // часть класса опущена

        return new ActivityMainBinding((ConstraintLayout) rootView, author, avatar, content, footer,
            header, like, menu, published, share);
    }
}
```



FINDVIEWBYID

Таким образом, `findViewById` никуда не девается, он просто генерируется автоматически.

Важно: мы специально смотрим сгенерированные исходники, чтобы вы понимали, как оно устроено внутри. Непонимание этого ведёт к:

1. Проблемам «не работает, не понятно почему»
2. Отсутствию понимания возможностей и ограничений

РАЗНЫЕ ТИПЫ

Стоит отметить, что если один и тот же id используется в разных layout'ах (например, в одном like — `ImageButton`, а в другом — `TextView`), то like в итоговом `ActivityMainBinding` будет типа `View` — т.е. типобезопасность сохраняется и вам самим придётся приводить тип (через `as` или `is`).



TOGGLE

Пока наше приложение работает достаточно странно: можно лишь залайкать пост, но не снять лайк. Хотя в большинстве социальных приложений вы можете это сделать.

Давайте подумаем, как мы можем это реализовать?



ДАННЫЕ

ДАННЫЕ

Хранить информацию о том, залайкан пост или нет просто в виде состояния (а именно изображения внутри `ImageButton`) — плохая идея.

Всегда нужно разделять отображение и данные. Поэтому, конечно же, данные нужно хранить в виде отдельного класса:

```
package ru.netology.nmedia.dto

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

```

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

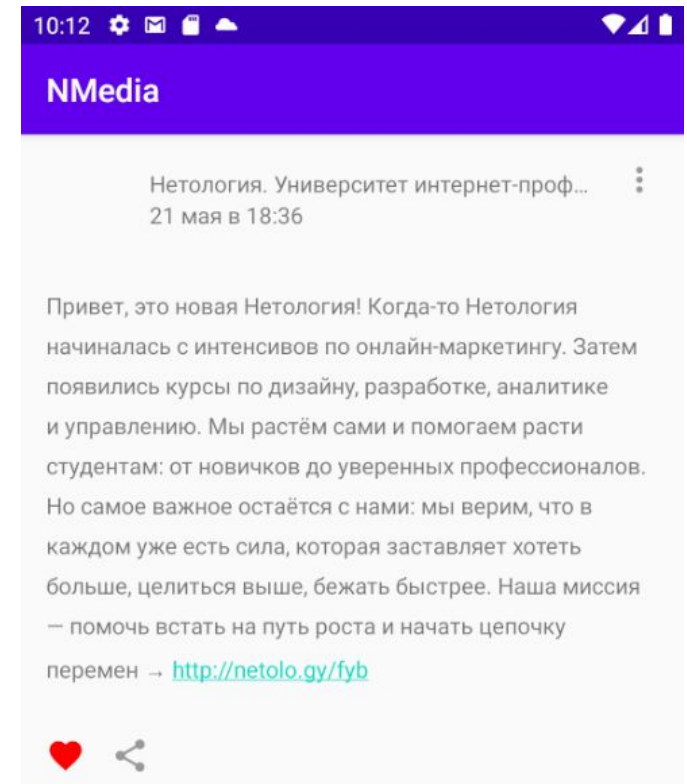
        val post = Post(
            id = 1,
            author = "Нетология. Университет интернет-профессий будущего",
            content = "Привет, это новая Нетология! Когда-то Нетология начиналась с интенс ...
            published = "21 мая в 18:36",
            likedByMe = false
        )
        with(binding) { this: ActivityMainBinding
            author.text = post.author
            published.text = post.published
            content.text = post.content
            if (post.likedByMe) {
                like?.setImageResource(R.drawable.ic_liked_24)
            }

            like?.setOnClickListener { it: View!
                post.likedByMe = !post.likedByMe
                like.setImageResource(
                    if (post.likedByMe) R.drawable.ic_liked_24 else R.drawable.ic_like_24
                )
            }
        } ^with
    }
}

```

ДАННЫЕ

Несмотря на то, что наше приложение ещё далеко от профессионального, оно уже работает, и мы добились решения поставленной задачи.





ИТОГИ



ИТОГИ

Сегодня мы обсудили базовые вопросы обработки событий и взаимодействия с интерфейсом.