Свежий взгляд на отображение диалогов в Android

* [Блог компании MobileUp](https://habr.com/ru/company/mobileup/),
* [Разработка мобильных приложений](https://habr.com/ru/hub/mobile_dev/),
* [Разработка под Android](https://habr.com/ru/hub/android_dev/)

На картинке первая мысль читателя, который недоумевает, что можно написать про такую простую задачу как отображения диалога. Аналогично думает и менеджер: «Тут ничего сложного, наш [Вася](https://habr.com/ru/users/Jeevuz/) за 5 минут сделает». Я, конечно, утрирую, но на самом деле всё не так просто, как кажется на первый взгляд. Особенно если мы говорим про Android.

Итак, на дворе шёл 2019 год, а **мы всё ещё не умеем нормально показывать диалоги**.

Давайте всё по порядку, и начнем с постановки задачи:

Требуется показать простой диалог с текстом для подтверждения действия и кнопками «подтвердить/отмена». По нажатию на кнопку «подтвердить» — совершить действие, по кнопке «отмена» — закрыть диалог.

Решение «в лоб»

Я бы назвал этот способ джуниорским, потому что не первый раз сталкиваюсь с непониманием, почему нельзя просто использовать AlertDialog, как показано ниже:

AlertDialog.Builder(this)

.setMessage("Please, confirm the action")

.setPositiveButton("Confirm") { dialog, which ->

*// handle click*

}

.setNegativeButton("Cancel", null)

.create()

.show()

Довольно распространенный способ для начинающего разработчика, он очевиден и интуитивно понятен. Но, как и во многих случаях при работе с Android, этот способ совершенно неправильный. На ровном месте мы получаем утечку памяти, достаточно повернуть устройство, и вы увидете в логах такую ошибку:

stacktrace

E/WindowManager: android.view.WindowLeaked: Activity com.example.testdialog.MainActivity has leaked window DecorView@71b5789[MainActivity] that was originally added here

at android.view.ViewRootImpl.<init>(ViewRootImpl.java:511)

at android.view.WindowManagerGlobal.addView(WindowManagerGlobal.java:346)

at android.view.WindowManagerImpl.addView(WindowManagerImpl.java:93)

at android.app.Dialog.show(Dialog.java:329)

at com.example.testdialog.MainActivity.onCreate(MainActivity.kt:27)

at android.app.Activity.performCreate(Activity.java:7144)

at android.app.Activity.performCreate(Activity.java:7135)

at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1271)

at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2931)

at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:3086)

at android.app.servertransaction.LaunchActivityItem.execute(LaunchActivityItem.java:78)

at android.app.servertransaction.TransactionExecutor.executeCallbacks(TransactionExecutor.java:108)

at android.app.servertransaction.TransactionExecutor.execute(TransactionExecutor.java:68)

at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1816)

at android.os.Handler.dispatchMessage(Handler.java:106)

at android.os.Looper.loop(Looper.java:193)

at android.app.ActivityThread.main(ActivityThread.java:6718)

at java.lang.reflect.Method.invoke(Native Method)

at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:493)

at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:858)

На Stackoverflow [вопрос](https://stackoverflow.com/questions/2850573/activity-has-leaked-window-that-was-originally-added) по этой проблеме один из самых популярных. Если коротко, то проблема в том, что мы либо показываем диалог, либо не закрываем диалог после завершения работы активити.

Можно, конечно, вызывать dismiss у диалога в onPause или onDestroy активити, как советуют в ответе по [ссылке](https://stackoverflow.com/a/2850597). Но это не совсем то, что нам нужно. Мы хотим, чтобы диалог восстанавливался после поворота устройства.

Устаревший способ

До появления фрагментов в Android диалоги должны были отображаться через вызов метода активити [showDialog](https://developer.android.com/reference/android/app/Activity.html" \l "showDialog(int,%20android.os.Bundle)). В этом случае активити правильно управляет жизненным циклом диалога и восстанавливает его после поворота. Создание самого диалога нужно было реализовать в коллбэке onCreateDialog:

public class MainActivity extends Activity {

private static final int CONFIRMATION\_DIALOG\_ID = 1;

*// ...*

@Override

protected Dialog onCreateDialog(int id, Bundle args) {

if (id == CONFIRMATION\_DIALOG\_ID) {

return new AlertDialog.Builder(this)

.setMessage("Please, confirm the action")

.setPositiveButton("Confirm", new DialogInterface.OnClickListener() {

@Override

public void onClick(DialogInterface dialog, int which) {

*// handle click*

}

})

.create();

} else {

return super.onCreateDialog(id, args);

}

}

}

Не очень удобно, что приходится заводить идентификатор диалога и передавать параметры через Bundle. И мы все ещё можем получить проблему «leaked window», если попытаемся отобразить диалог после вызова onDestroy у активити. Такое возможно, например, при попытке показать ошибку после асинхронной операции.

Вообще, эта проблема типична для Android, когда нужно что-то сделать после асинхронной операции, а активити или фрагмент уже уничтожен в этот момент. Наверное, поэтому MV\*-паттерны более популярны в Android-сообществе, чем среди iOS-разработчиков.

Способ из документации

В [Android Honeycomb](https://developer.android.com/about/versions/android-3.0-highlights) появились фрагменты, и описанный выше способ устарел, а метод [showDialog](https://developer.android.com/reference/android/app/Activity.html" \l "showDialog(int,%20android.os.Bundle)) у активити помечен как deprecated. Нет, AlertDialog не устарел, как ошибаются многие. Просто теперь появился [DialogFragment](https://developer.android.com/reference/android/app/DialogFragment), который оборачивает объект диалога и управляет его жизненным циклом.

Родные фрагменты тоже устарели начиная с 28 API. Теперь следует использовать только реализацию из Support Library(AndroidX).

Давайте реализуем нашу задачу, как это предписывает [официальная документация](https://developer.android.com/guide/topics/ui/dialogs):

1. Для начала нужно наследоваться от DialogFragment и реализовать создание диалога в методе onCreateDialog.
2. Описать интерфейс событий диалога и инстанцировать слушатель в методе onAttach.
3. Реализовать интерфейс событий диалога в активити или фрагменте.

Если читателю не очень понятно, почему нельзя передавать слушатель через конструктор, то он может почитать подробнее об этом [тут](https://stackoverflow.com/questions/12062946/why-do-i-want-to-avoid-non-default-constructors-in-fragments/16042750#16042750)

Код фрагмента диалога:

class ConfirmationDialogFragment : DialogFragment() {

interface ConfirmationListener {

fun confirmButtonClicked()

fun cancelButtonClicked()

}

private lateinit var listener: ConfirmationListener

override fun onAttach(context: Context?) {

super.onAttach(context)

try {

*// Instantiate the ConfirmationListener so we can send events to the host*

listener = activity as ConfirmationListener

} catch (e: ClassCastException) {

*// The activity doesn't implement the interface, throw exception*

throw ClassCastException(activity.toString() + " must implement ConfirmationListener")

}

}

override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {

return AlertDialog.Builder(context!!)

.setMessage("Please, confirm the action")

.setPositiveButton("Confirm") { \_, \_ ->

listener.confirmButtonClicked()

}

.setNegativeButton("Cancel") { \_, \_ ->

listener.cancelButtonClicked()

}

.create()

}

}

Код активити:

class MainActivity : AppCompatActivity(), ConfirmationListener {

private fun showConfirmationDialog() {

ConfirmationDialogFragment()

.show(supportFragmentManager, "ConfirmationDialogFragmentTag")

}

override fun confirmButtonClicked() {

*// handle click*

}

override fun cancelButtonClicked() {

*// handle click*

}

}

Достаточно много кода получилось, не так ли?

Как правило, в проекте есть какой-нибудь MVP, но я решил, что вызовы презентера можно опустить в данном случае. В примере выше стоит ещё добавить статический метод создания диалога newInstance и передачу параметров в аргументы фрагмента, всё как полагается.

И это всё ради того, чтобы диалог вовремя скрывался и правильно восстанавливался. Не удивительно, что появляются такие вопросы на Stackoverflow: [один](https://stackoverflow.com/questions/7977392/android-dialogfragment-vs-dialog) и [два](https://stackoverflow.com/questions/13765127/dialogfragment-advantages-over-alertdialog/13765411).

Поиск идеального решения

Текущее положение дел нас не устраивало, и мы стали искать способ, как сделать работу с диалогами более комфортной. Было ощущение, что можно сделать проще, почти как в первом способе.

Ниже сформулированы соображения, которыми мы руководствовались:

* **Нужно ли сохранять и восстанавливать диалог после убийства процесса приложения?**  
  В большинстве случаев это не требуется, как и в нашем примере, когда нужно показать простое сообщение или что-то спросить. Такой диалог актуален пока не потеряно внимание пользователя. Если его восстановить после долгого отсутствия в приложении, то пользователь потеряет контекст с планируемым действием. Поэтому **нужно только поддержать повороты устройства** и правильно обрабатывать жизненный цикл диалога. Иначе от неловкого движения устройства пользователь может потерять только что открытое сообщение, не прочитав его.
* При использовании DialogFragment появляется слишком много boilerplate-кода, теряется простота. Поэтому было бы неплохо избавиться от фрагмента как обёртки и **использовать Dialog напрямую**. Для этого придется хранить состояние диалога, чтобы показать его вновь после пересоздания View и скрывать, когда View умирает.
* Все привыкли воспринимать показ диалога как команду, особенно если работаешь только с MVP. Задачу последующего восстановление состояния берет на себя FragmentManager. Но можно посмотреть на эту ситуацию иначе и начать **воспринимать диалог как state**. Это намного удобнее при работе с паттернами PM или MVVM.
* Учитывая, что большинство приложений сейчас используют реактивные подходы, появляется потребность в том, **чтобы диалоги были реактивными**. Основная задача — не разрывать цепочку, которая инициирует показ диалога, и привязать реактивный поток событий для получения результата от него. Это очень удобно на стороне PresentationModel/ViewModel, когда манипулируешь несколькими потоками данных.

Мы учли все вышеописанные требования и придумали способ реактивного показа диалогов, который успешно реализовали в нашей библиотеке [RxPM](https://github.com/dmdevgo/RxPM) (про нее есть отдельная [статья](https://habr.com/ru/company/mobileup/blog/342850/)).

Само решение не требует библиотеки и может быть сделано отдельно. Руководствуясь идеей «диалог как state» можно попробовать построить решение на основе модных ViewModel и LiveData. Но я оставлю это право за читателем, а далее речь пойдет уже о готовом решении из библиотеки.

Реактивный способ

Я покажу, как исходная задача решается в RxPM, но сначала пару слов о ключевых понятиях из библиотеки:

* **PresentationModel** — хранит реактивный стейт, содержит UI-логику, переживает повороты.
* **State** — реактивный стейт. Можно воспринимать как обертку над BehaviorRelay.
* **Action** — обертка над PublishRelay, служит для передачи событий от View в PresentationModel.
* **State** и **Action** имеют observable и consumer.

За состояние диалога отвечает класс [DialogControl](https://github.com/dmdevgo/RxPM/blob/develop/rxpm/src/main/kotlin/me/dmdev/rxpm/widget/DialogControl.kt). Он имеет два параметра: первый для типа данных, которые должны отображаться в диалоге, второй — для типа результата. В нашем примере тип данных будет Unit, но это может быть сообщение пользователю или любой другой тип.

В DialogControl есть следующие методы:

* show(data: T) — просто отдает команду на отображение.
* showForResult(data: T): Maybe<R> — показывает диалог и открывает поток для получения результата.
* sendResult(result: R) — отправляет результат, вызывается со стороны View.
* dismiss() — просто скрывает диалог.

В DialogControl хранится состояние — есть диалог на экране или нет (Displayed/Absent). Вот так это выглядит в коде класса:

class DialogControl<T, R> internal constructor(pm: PresentationModel) {

val displayed = pm.State<Display>(Absent)

private val result = pm.Action<R>()

sealed class Display {

data class Displayed<T>(val data: T) : Display()

object Absent : Display()

}

*// ...*

}

Создадим простую PresentationModel:

class SamplePresentationModel : PresentationModel() {

enum class ConfirmationDialogResult {

CONFIRMED, CANCELED

}

*// Создаем контрол диалога без данных и с enum для возвращаемого результата*

val confirmationDialog = dialogControl<Unit, ConfirmationDialogResult>()

val buttonClicks = Action<Unit>()

override fun onCreate() {

super.onCreate()

buttonClicks.observable

.switchMapMaybe {

*// по клику на кнопку запускаем диалог и ожидаем нужный результат*

confirmationDialog.showForResult(Unit)

.filter { it == ConfirmationDialogResult.CONFIRMED }

}

.subscribe {

*// обрабатываем действие*

}

.untilDestroy()

}

}

Обратите внимание, что обработка кликов, получение подтверждения и обработка действия реализованы в одной цепочке. Это позволяет сделать код сфокусированным и не раскидывать логику по нескольким коллбэкам.

Далее просто привязываем DialogControl во View с помощью экстеншена bindTo.  
Собираем обычный AlertDialog, а результат отправляем через sendResult:

class SampleActivity : PmSupportActivity<SamplePresentationModel>() {

override fun providePresentationModel() = SamplePresentationModel()

*// В этом методе связываем View и PresentationModel*

override fun onBindPresentationModel(pm: SamplePresentationModel) {

pm.confirmationDialog bindTo { data, dialogControl ->

AlertDialog.Builder(this@SampleActivity)

.setMessage("Please, confirm the action")

.setPositiveButton("Confirm") { \_, \_ ->

dialogControl.sendResult(CONFIRMED)

}

.setNegativeButton("Cancel") { \_, \_ ->

dialogControl.sendResult(CANCELED)

}

.create()

}

button.clicks() bindTo pm.buttonClicks

}

}

При типичном сценарии под капотом происходит примерно следующее:

1. Кликаем на кнопку, событие через Action «buttonClicks» попадает в PresentationModel.
2. По этому событию запускаем отображение диалога через вызов showForResult.
3. В результате состояние в DialogControl меняется с Absent на Displayed.
4. При получении события Displayed — вызывается лямбда, которую мы передали в привязке bindTo. В ней создается объект диалога, который затем показывается.
5. Пользователь нажимает, кнопку «Confirm», срабатывает слушатель и результат нажатия отправляется в DialogControl посредством вызова sendResult.
6. Далее результат попадает во внутренний Action «result», а состояние с Displayed меняется на Absent.
7. При получении события Absent текущий диалог закрывается.
8. Событие от Action «result» попадает в поток, который был открыт вызовом showForResult и обрабатывается цепочкой в PresentationModel.

Стоит отметить, что диалог закрывается и в момент, когда View отвязывается от PresentationModel. В этом случае состояние остается Displayed. Оно будет получено при следующей привязке и диалог будет восстановлен.

Как видите, необходимость в DialogFragment пропала. Диалог показывается, когда View привязывается к PresentationModel и скрывается, когда View отвязывается. За счёт того, что состояние хранится в DialogControl, который в свою очередь хранится в PresentationModel, диалог восстанавливается после поворота устройства.

Пишите диалоги правильно

Мы с вами рассмотрели несколько способов отображения диалогов. Если вы все ещё показываете первым способом, то прошу вас, не делайте больше так. Для любителей MVP ничего не остается, как использовать стандартный способ, который описан в официальной документации. К сожалению, склонность к императивности этого паттерна не позволяет сделать по-другому. Ну, а фанатам RxJava рекомендую присмотреться к реактивному способу и нашей библиотеке [RxPM](https://github.com/dmdevgo/RxPM).