[Урок 4 ViewModel](https://startandroid.ru/ru/courses/architecture-components/27-course/architecture-components/527-urok-4-viewmodel.html" \o "Урок 4. ViewModel)

В этом уроке рассмотрим, как использовать ViewModel. Как сохранять данные при повороте экрана. Как передать Context в ViewModel. Как передать свои данные в конструктор модели с помощью фабрики. Как передать данные между фрагментами. Что использовать: ViewModel или onSavedInstanceState.

[ViewModel](https://developer.android.com/reference/android/arch/lifecycle/ViewModel.html) - класс, позволяющий Activity и фрагментам сохранять необходимые им объекты живыми при повороте экрана.

Давайте сразу посмотрим пример и по ходу дела разберем нюансы и прочую теорию. Как подключить библиотеку к проекту, вы можете посмотреть в начале [Урока 2](https://startandroid.ru/ru/courses/architecture-components/27-course/architecture-components/525-urok-2-livedata.html).

Создаем свой класс, наследующий ViewModel

|  |  |
| --- | --- |
| 1  2  3 | public class MyViewModel extends ViewModel {    } |

Пока оставим его пустым.

Чтобы добраться до него в Activity, нужен следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Override  protected void onCreate(Bundle savedInstanceState) {     super.onCreate(savedInstanceState);     setContentView(R.layout.activity\_main);       MyViewModel model = ViewModelProviders.of(this).get(MyViewModel.class);       // ...  } |

В метод [ViewModelProviders.of](https://developer.android.com/reference/android/arch/lifecycle/ViewModelProviders.html" \l "of(android.support.v4.app.FragmentActivity)" \t "_blank) передаем Activity. Тем самым мы получим доступ к провайдеру, который хранит все ViewModel для этого Activity.

Методом [get](https://developer.android.com/reference/android/arch/lifecycle/ViewModelProvider.html" \l "get(java.lang.Class%3CT%3E)" \t "_blank) запрашиваем у этого провайдера конкретную модель по имени класса - MyViewModel. Если провайдер еще не создавал такой объект ранее, то он его создает и возвращает нам. И пока Activity окончательно не будет закрыто, при всех последующих вызовах метода get мы будем получать этот же самый объект MyViewModel.

Соответственно, при поворотах экрана, Activity будет пересоздаваться, а объект MyViewModel будет спокойно себе жить в провайдере. И Activity после пересоздания сможет получить этот объект обратно и продолжить работу, как будто ничего не произошло.

Отсюда следует важный вывод. Не храните в ViewModel ссылки на Activity, фрагменты, View и пр. Это может привести к утечкам памяти.

На картинке время жизни (оно же scope) модели это выглядит так:

Модель жива, пока Activity не закроется окончательно.

У метода get, который возвращает нам модель из провайдера, есть еще такой вариант вызова:

[T get (String key, Class<T> modelClass)](https://developer.android.com/reference/android/arch/lifecycle/ViewModelProvider.html#get(java.lang.String,%20java.lang.Class%3CT%3E))

Т.е. вы можете создавать несколько моделей одного и того же класса, но использовать разные текстовые ключи для их хранения в провайдере.

## LiveData

LiveData очень удобно использовать с ViewModel. В прошлых уроках я для работы с LiveData использовал синглтон, но теперь мы перейдем на ViewModel.

Рассмотрим несложный пример асинхронной однократной загрузки каких-либо данных:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class MyViewModel extends ViewModel {       // ...       MutableLiveData<String> data;       public LiveData<String> getData() {         if (data == null) {             data = new MutableLiveData<>();             loadData();         }         return data;     }       private void loadData() {         dataRepository.loadData(new Callback<String>() {             @Override             public void onLoad(String s) {                 data.postValue(s);             }         });     }    } |

Основной метод здесь - это getData. Когда Activity захочет получить данные, оно вызовет именно этот метод. Мы проверяем, создан ли уже MutableLiveData. Если нет, значит этот метод вызывается первый раз. В этом случае создаем MutableLiveData и стартуем асинхронный процесс получения данных методом loadData. Далее возвращаем LiveData.

В методе loadData происходит асинхронное получение данных из какого-нибудь репозитория. Как только данные будут получены (в методе onLoad), мы передаем их в MutableLiveData.

Метод loadData должен быть асинхронным, потому что он вызывается из метода getData, а getData в свою очередь вызывается из Activity и все это происходит в UI потоке. Если loadData начнет грузить данные синхронно, то он заблокирует UI поток.

Код в Activity выглядит так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @Override  protected void onCreate(Bundle savedInstanceState) {     super.onCreate(savedInstanceState);     setContentView(R.layout.activity\_main);       MyViewModel model = ViewModelProviders.of(this).get(MyViewModel.class);     LiveData<String> data = model.getData();     data.observe(this, new Observer<String>() {         @Override         public void onChanged(@Nullable String s) {             // ...         }     });  } |

Получаем от провайдера модель. От модели получаем LiveData, на который подписываемся и ждем данные.

В этом примере ViewModel нужен, чтобы сохранить процесс получения данных при повороте экрана. А LiveData - для удобного асинхронного получения данных.

Т.е. это будет выглядеть так:

- Activity вызывает метод модели getData  
- модель создает MutableLiveData и стартует асинхронный процесс получения данных от репозитория  
- Activity подписывается на полученный от модели LiveData и ждет данные  
- происходит поворот экрана  
- на модели этот поворот никак не сказывается, она спокойно сидит в провайдере и ждет ответ от репозитория  
- Activity пересоздается, получает ту же самую модель от провайдера, получает тот же самый LiveData от модели и подписывается на него и ждет данные  
- репозиторий возвращает данные, модель передает их в MutableLiveData   
- Activity получает данные данные от LiveData

Если репозиторий вдруг пришлет ответ в тот момент, когда Activity будет пересоздаваться, то Activity получит этот ответ, как только подпишется на LiveData.

Если ваш репозиторий сам умеет возвращать LiveData, то все значительно упрощается. Вы просто отдаете этот LiveData в Activity и оно подписывается.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class MyViewModel extends ViewModel {        // ...        LiveData<String> data;        public LiveData<String> getData() {          if (data == null) {              data = dataRepository.loadData();          }          return data;      }  } |

## Очистка ресурсов

Когда Activity окончательно закрывается, провайдер удаляет ViewModel, предварительно вызвав его метод [onCleared](https://developer.android.com/reference/android/arch/lifecycle/ViewModel.html" \l "onCleared()" \t "_blank)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class MyViewModel extends ViewModel {       // ...       @Override     protected void onCleared() {         // clean up resources       }  } |

В этом методе вы сможете выполнить все необходимые операции по освобождению ресурсов, закрытию соединений/потоков и т.п.

## Context

Не стоит передавать Activity в модель в качестве Context. Это может привести к утечкам памяти.

Если вам в модели понадобился объект Context, то вы можете наследовать не ViewModel, а [AndroidViewModel](https://developer.android.com/reference/android/arch/lifecycle/AndroidViewModel.html" \t "_blank).

|  |
| --- |
| public class MyViewModel extends AndroidViewModel {       public MyViewModel(@NonNull Application application) {         super(application);     }       public void doSomething() {         Context context = getApplication();         // ....     }    } |

При создании этой модели, провайдер передаст ей в конструктор класс Application, который является Context. Вы сможете до него добраться методом getApplication.

Код получения этой модели в Activity останется тем же самым.

## Передача объектов в конструктор модели

Бывает необходимость передать модели какие-либо данные при создании. Модель создается провайдером и у нас есть возможность вмешаться в этот процесс. Для этого используется фабрика. Мы учим эту фабрику создавать модель так, как нам нужно. И провайдер воспользуется этой фабрикой, когда ему понадобится создать объект.

Рассмотрим пример. У нас есть такая модель

|  |
| --- |
| public class MyViewModel extends ViewModel {       private final long id;       public MyViewModel(long id) {         this.id = id;     }       // ...    } |

Ей нужен long при создании.

Создаем фабрику

|  |
| --- |
| public class ModelFactory extends ViewModelProvider.NewInstanceFactory {       private final long id;       public ModelFactory(long id) {         super();         this.id = id;     }       @NonNull     @Override     public <T extends ViewModel> T create(@NonNull Class<T> modelClass) {         if (modelClass == MyViewModel.class) {             return (T) new MyViewModel(id);         }         return null;     }  } |

Она должна наследовать класс [ViewModelProvider.NewInstanceFactory](https://developer.android.com/reference/android/arch/lifecycle/ViewModelProvider.NewInstanceFactory.html" \t "_blank).

В конструктор передаем long, который нам необходимо будет передать в модель.

В методе [create](https://developer.android.com/reference/android/arch/lifecycle/ViewModelProvider.NewInstanceFactory.html" \l "create(java.lang.Class%3CT%3E)" \t "_blank) фабрика получит от провайдера на вход класс модели, которую необходимо создать. Проверяем, что это класс MyViewModel, сами создаем модель и передаем туда long.

В Activity код получения модели будет выглядеть так:

|  |  |
| --- | --- |
|  | long id = ...;    MyViewModel model = ViewModelProviders.of(this, new ModelFactory(id))         .get(MyViewModel.class); |

Мы создаем новую фабрику с нужными нам данными и передаем ее в метод of. При вызове метода get провайдер использует фабрику для создания модели, т.е. выполнится наш код создания модели и передачи в нее данных.

## Передача данных между фрагментами

ViewModel может быть использована для передачи данных между фрагментами, которые находятся в одном Activity. В документации есть отличный пример кода:

|  |
| --- |
| public class SharedViewModel extends ViewModel {      private final MutableLiveData<Item> selected = new MutableLiveData<Item>();        public void select(Item item) {          selected.setValue(item);      }        public LiveData<Item> getSelected() {          return selected;      }  } |

SharedViewModel - модель с двумя методами: один позволяет поместить данные в LiveData, другой - позволяет получить этот LiveData. Соответственно, если два фрагмента будут иметь доступ к этой модели, то один сможет помещать данные в его LiveData, а другой - подпишется и будет получать эти данные. Таким образом два фрагмента будут обмениваться данными ничего не зная друг о друге.

Чтобы два фрагмента могли работать с одной и той же моделью, они могут использовать общее Activity. Код получения модели в фрагментах выглядит так:

|  |  |
| --- | --- |
| 1 | SharedViewModel model = ViewModelProviders.of(getActivity()).get(SharedViewModel.class); |

Для обоих фрагментов getActivity вернет одно и то же Activity. Метод ViewModelProviders.of вернет провайдера этого Activity. Далее методом get получаем модель.

Код фрагментов:

|  |
| --- |
| public class MasterFragment extends Fragment {      private SharedViewModel model;      public void onCreate(Bundle savedInstanceState) {          super.onCreate(savedInstanceState);          model = ViewModelProviders.of(getActivity()).get(SharedViewModel.class);          itemSelector.setOnClickListener(item -> {              model.select(item);          });      }  }    public class DetailFragment extends Fragment {      public void onCreate(Bundle savedInstanceState) {          super.onCreate(savedInstanceState);          SharedViewModel model = ViewModelProviders.of(getActivity()).get(SharedViewModel.class);          model.getSelected().observe(this, { item ->             // Update the UI.          });      }  } |

Фрагмент MasterFragment помещает данные в LiveData. А DetailFragment - подписывается и получает данные.

## onSavedInstanceState

Чем ViewModel отличается от onSavedInstanceState. Для каких данных какой из них лучше использовать. Кажется, что если есть ViewModel, который жив все время, пока не закрыто Activity, то можно забыть про onSavedInstanceState. Но это не так.

Давайте в качестве примера рассмотрим Activity, которое отображает список каких-то данных и может выполнять поиск по ним. Пользователь открывает Activity и выполняет поиск. Activity отображает результаты этого поиска. Пользователь сворачивает приложение. Когда он его снова откроет, он ожидает, что там все останется в этом же состоянии.

Но тут внезапно системе не хватает памяти и она убивает это свернутое приложение. Когда пользователь снова запустит его, Activity ничего не будет знать о поиске, и просто покажет все данные. В этом случае ViewModel нам никак не поможет, потому что модель будет убита вместе с приложением. А вот onSavedInstanceState будет выполнен. В нем мы сможем сохранить поисковый запрос, и при последующем запуске получить его из объекта savedInstanceState и выполнить поиск. В результате пользователь увидит тот же экран, который был, когда приложение было свернуто.

Итак.

ViewModel - здесь удобно держать все данные, которые нужны вам для формирования экрана. Они будут жить при поворотах экрана, но умрут, когда приложение будет убито системой.

onSavedInstanceState - здесь нужно хранить тот минимум данных, который понадобится вам для восстановления состояния экрана и данных в ViewModel после экстренного закрытия Activity системой. Это может быть поисковый запрос, ID и т.п.

Соответственно, когда вы достаете данные из savedInstanceState и предлагаете их модели, это может быть в двух случаях:

1) Был обычный поворот экрана. В этом случае ваша модель должна понять, что ей эти данные не нужны, потому что при повороте экрана модель ничего не потеряла. И уж точно модель не должна заново делать запросы в БД, на сервер и т.п.

2) Приложение было убито, и теперь запущено заново. В этом случае модель берет данные из savedInstanceState и использует их, чтобы восстановить свои данные. Например, берет ID и идет в БД за полными данными.

## RxJava

В начале этого урока мы рассмотрели пример работы ViewModel и LiveData. Возникает вопрос, можно ли заменить LiveData на Flowable?

У LiveData есть одно большое преимущество - он учитывает состояние Activity. Т.е. он не будет слать данные, если Activity свернуто. И он отпишет от себя Activity, которое закрывается.

А вот Flowable этого не умеет. Если в модели есть Flowable, и Activity подпишется на него, то этот Flowable будет держать Activity, пока оно само явно не отпишется (или пока Flowable не завершится).

Давайте рассмотрим пример. ViewModel обычно работает с репозиториями, которые могут быть синглтонами. В репозитории есть какой-то объект для подписки (типа LiveData или Flowable). Репозиторий периодически обновляет в нем данные. Модель берет этот объект из репозитория и отдает его в Activity, и Activity подписывается на этот объект. Объект теперь хранит ссылку на Activity.

Таким образом получилось, что репозиторий держит ссылку на Activity через объект подписки. И если мы закроем Activity, но не отпишем его от объекта подписки, то возникнет утечка памяти, т.к. репозиторий может жить все время работы приложения. И все это время Activity будет висеть в памяти.

Давайте рассмотрим, как это решается в случае с LiveData или Flowable. Важно понимать, что будет происходить с подпиской при закрытии Activity. ViewModel будем рассматривать только как инструмент передачи объекта из репозитория в Activity.

1) ViewModel готов из репозитория предоставить LiveData. И мы в Activiy хотели бы работать с LiveData.

Цепочка ссылок:  
Repository -> LiveData -> Activity

Тут получается полная идилия. Activity берет из модели LiveData, подписывается на него и все ок. При закрытии Activity не будет никаких утечек памяти и прочих проблем с подпиской, т.к. LiveData сам отпишет Activity и тем самым разорвет цепочку ссылок.

2) ViewModel готов вернуть нам Flowable, а мы в Activity хотели бы работать с LiveData.

В этом случае конвертируем Flowable в LiveData внутри модели и отдаем LiveData в Activity.

Цепочка ссылок:  
Repository -> Flowable -> LiveData -> Activity

Activity снова будет подписано на LiveData. А это значит, что нам, как и в первом варианте, не надо заботиться об этой подписке. LiveData отпишет от себя Activity, и сам отпишется от Flowable. Цепочка ссылок разорвется в двух местах.

3) ViewModel готов вернуть нам LiveData, а мы в Activity хотели бы работать с Flowable.

В этом случае передаем LiveData в Activity и преобразуем его в Flowable.

Цепочка ссылок:  
Repository -> LiveData -> Flowable -> Activity

Activity будет подписано на Flowable. А Flowable будет подписан на LiveData. При этом подписка Flowable на LiveData будет работать с учетом Activity LifeCycle. И когда Activity будет закрыто, LiveData сам отпишет от себя Flowable.

Цепочка ссылок разорвется, но, в любом случае, хорошей практикой является отписка от Flowable вручную при закрытии Activity.

4) ViewModel готов вернуть нам Flowable, и мы в Activity хотели бы работать с Flowable.

В этом случае Activity будет подписано на Flowable.

Цепочка ссылок:  
Repository -> Flowable -> Activity

При закрытии Activity нам самим необходимо отписать Activity от Flowable.

Как конвертировать LiveData во Flowable и наоборот, мы рассматривали в [Уроке 3](https://startandroid.ru/ru/courses/architecture-components/27-course/architecture-components/526-urok-3-livedata.html).