№11 Архитектура приложения. Room и LiveData.

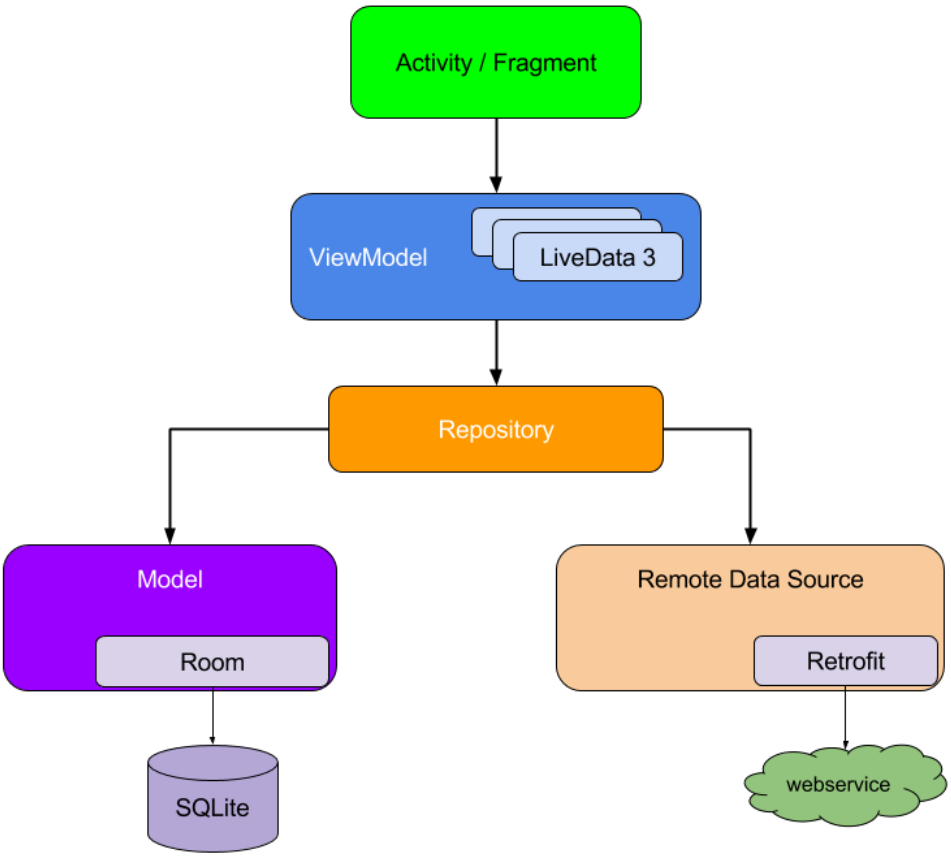
Архитектурные компоненты

Компоненты архитектуры помогают структурировать приложение таким образом, чтобы оно было надежным, проверяемым и поддерживаемым с меньшим количеством шаблонов.

https://developer.android.com/jetpack/getting-started

Архитектура фокусируется на подмножестве компонентов, а именно LiveData, ViewModel и Room.

https://developer.android.com/topic/libraries/architecture



Разберемся с типами.

Entity. Используем при работе с компонентами архитектуры. Это аннотированный класс, который описывает таблицу базы данных.

База данных SQLite: Библиотека сохранения состояний создает и поддерживает базу данных.

DAO: объект доступа к данным. Отображение SQL-запросов на функции. Мы определяли их в своем классе SQLiteOpenHelper. Когда вы используете DAO, вы вызываете методы, и Room заботится обо всем остальном.

Room: держатель базы данных, который служит точкой доступа к базе данных SQLite, использует DAO для запросов.

Репозиторий: класс, который вы создаете для управления несколькими источниками данных.

ViewModel: предоставляет данные для пользовательского интерфейса. Действует как центр связи между репозиторием и пользовательским интерфейсом. Скрывает, откуда данные берутся из пользовательского интерфейса.

LiveData: класс хранителя данных, который можно наблюдать. Всегда хранит/кэширует последнюю версию данных. Уведомляет своих наблюдателей, когда данные были изменены.

Итак, **Room** предоставляет слой абстракции над **SQLite**, чтобы обеспечить свободный доступ к базе данных SQLite.

Наиболее распространенным вариантом использования Room является кэширование фрагментов данных. Когда устройство не может получить доступ к сети, пользователь может просматривать контент, находясь в автономном режиме. Любые пользовательские изменения контента затем синхронизируются с сервером после того, как устройство снова подключится к сети.

Google настоятельно рекомендует использовать **Room**.

https://codelabs.developers.google.com/codelabs/android-room-with-a-view-kotlin/#0

Однако, для его использования понадобятся еще некоторые компоненты.

Lifecycle

https://developer.android.com/reference/androidx/lifecycle/Lifecycle

Довольно часто часть логики приложения завязана на жизненный цикл Activity. Мы включаем что-либо в методах **onStart** или **onResume** и выключаем в **onPause** или **onStop**.

Например. Есть какой-то класс для работы с сервером. Он должен взаимодействовать с сервером, пока Activity открыта. Соответственно, мы будем подключать его к серверу при показе Activity и отключать при скрытии Activity.

```
public class MyServer {
    public void connect() {
        // ...
    }
    public void disconnect() {
        // ...
    }
}
```

Метод **connect** используется для подключения к серверу, **disconnect** - для отключения.

Вызываем эти методы в **onStart** и **onStop** в Activity.

```
@Override
protected void onStart() {
    super.onStart();
    myServer.connect();
}

@Override
protected void onStop() {
    super.onStop();
    myServer.disconnect();
}
```

Это вполне классическая, часто используемая схема. И в простом примере все выглядит понятно. Но в сложных приложениях содержание методов **onStart**, **onStop** и пр. может состоять из нескольких десятков строк и быть достаточно запутанным. Для улутшения архитектуры Google рекомендует выносить эту логику из Activity.

Это можно сделать так. У Activity есть метод **getLifecycle**, который возвращает объект **Lifecycle**. На этот объект можно подписать слушателей, которые будут получать уведомления при смене lifecycle-состояния Activity.

Activity и Fragment в Support Library, начиная с версии 26.1.0 реализуют интерфейс **LifecycleOwner**. Именно этот интерфейс и добавляет им метод **getLifecycle**.

У вас должна быть такая строка в **build.gradle** файле модуля, в секции dependencies

```
implementation 'androidx.appcompat:appcompat:1.2.0'
```

В примере слушателем будет **MyServer**. Чтобы иметь возможность подписаться на **Lifecycle**, он должен реализовывать интерфейс **LifecycleObserver**.

```
public class MyServer implements LifecycleObserver {

    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    public void connect() {
        // ...
    }
    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
    public void disconnect() {
        // ...
    }
}
```

Обратите внимание, что интерфейс **LifecycleObserver** пустой. В нем нет кучи методов типа **onStart**, **onStop** и т.п. Мы просто помечаем в классе MyServer его же собственные методы аннотацией **OnLifecycleEvent** и указываем, при каком lifecycle-событии метод должен быть вызван.

В примере, мы указываем, что метод **connect** должен вызываться в момент о**nStart**, а метод **disconnect** - в момент **onStop**

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    // ...
    getLifecycle().addObserver(myServer);
}
```

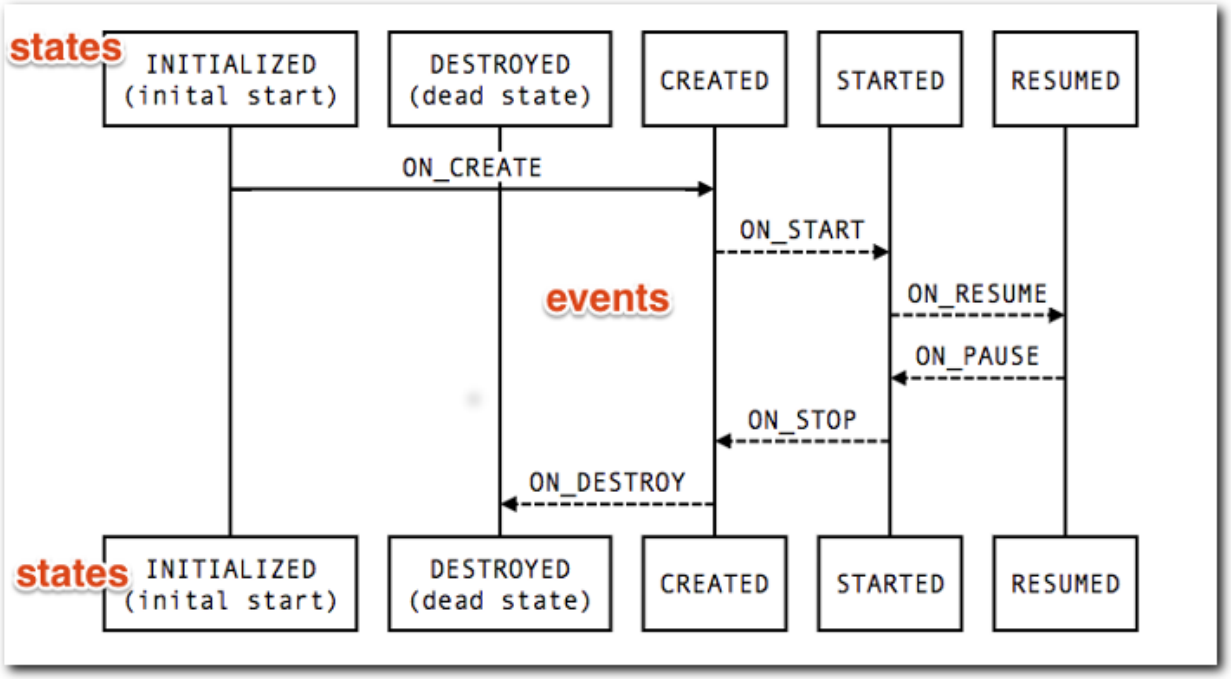
В Activity методом **getLifecycle** получаем **Lifecycle**, и методом **addObserver** подписываем **myServer**.

А методы **onStart** и **onStop** в Activity нам больше не нужны, их можно удалить.

Теперь, при переходе Activity из состояния **CREATED** в состояние **STARTED**, его объект **Lifecycle** вызовет метод *myServer.connect*. А при переходе из **STARTED** в **CREATED** - Lifecycle вызовет *myServer.disconnect*.

При этом в Acivity это потребовало от нас минимум кода - только подписать myServer на Lifecycle. Все остальное решает сам MyServer.

На схеме ниже вы можете увидеть какие состояние проходит Activity и какие события при этом вызываются.



Ничего нового тут для вас нет. Тут вы можете видеть состояния и события. При переходе между состояниями происходят события.

Эти события мы указывали в аннотациях *OnLifecycleEvent* к методам объекта MyServer.

Полный список событий можно посмотреть в документации.

Отписаться от Lifecycle можно методом r**emoveObserver**.

Any

Вы можете использовать событие **ON_ANY** для получения всех событий в одном методе

```
    @OnLifecycleEvent(ON_ANY)
    void onAny(LifecycleOwner source, Lifecycle.Event event) {
        // ...
```

В этом случае все события будут вызывать этот метод.

Используйте входящий параметр **event**, чтобы определить, какое именно событие произошло.

Состояние

Если вы хотите узнать текущее состояние Activity, то у его объекта **Lifecycle** есть метод **getCurrentState**:

```
if (getLifecycle().getCurrentState() == Lifecycle.State.RESUMED) {
        // ...
    }
```

Также, вы можете проверить, что текущее состояние Activity не ниже определенного состояния.

```
if (getLifecycle().getCurrentState().isAtLeast(Lifecycle.State.STARTED)) {
        // ...
    }
```

Метод **isAtLeast** проверяет, что состояние Activity не ниже, чем STARTED. Т.е. либо STARTED, либо RESUMED.

LiveData

https://developer.android.com/topic/libraries/architecture/livedata

В **build.gradle** файле модуля добавьте dependencies:

```
dependencies {
    def lifecycle_version = "2.2.0"
    def arch_version = "2.1.0"

    // ViewModel
    implementation "androidx.lifecycle:lifecycle-viewmodel:$lifecycle_version"
    // LiveData
    implementation "androidx.lifecycle:lifecycle-livedata:$lifecycle_version"
    // Lifecycles only (without ViewModel or LiveData)
    implementation "androidx.lifecycle:lifecycle-runtime:$lifecycle_version"
// optional - ReactiveStreams support for LiveData
    implementation "androidx.lifecycle:lifecycle-reactivestreams:$lifecycle_version"
```

LiveData - хранилище данных, работающее по принципу паттерна Observer (наблюдатель). Это хранилище умеет делать две вещи:

1) в него можно поместить какой-либо объект

2) на него можно подписаться и получать объекты, которые в него помещают.

Т.е. с одной стороны кто-то помещает объект в хранилище, а с другой стороны кто-то подписывается и получает этот объект.

В качестве аналогии можно привести, например, каналы в Telegram. Автор пишет пост и отправляет его в канал, а все подписчики получают этот пост.

Казалось бы, ничего особо в таком хранилище нет, но есть один очень важный нюанс. **LiveData** умеет определять активен подписчик или нет, и *отправлять данные будет только активным подписчикам*. Предполагается, что подписчиками **LiveData** будут **Activity** и фрагменты. А их состояние активности будет определяться с помощью их **Lifecycle** объекта.

Получение данных из LiveData

Пусть у нас есть некий синглтон класс **DataController** из которого можно получить **LiveData**<**String**>.

```
LiveData<String> liveData = DataController.getInstance().getData();
```

DataController периодически что-то там внутри себя делает и обновляет данные в **LiveData**.

Посмотрим, как Activity может подписаться на **LiveData** и получать данные, которые помещает в него **DataController**. Код в Activity будет выглядеть так:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    LiveData<String> liveData = DataController.getInstance().getData();

    liveData.observe(this, new Observer<String>() {
        @Override
        public void onChanged(@Nullable String value) {
            textView.setText(value)
        }
    });
}
```

Получаем **LiveData** из **DataController**, и методом **observe** подписываемся. В метод **observe** нам необходимо передать два параметра:

1) это **LifecycleOwner**. Activity и фрагменты в Support Library, начиная с версии 26.1.0 реализуют этот интерфейс, поэтому мы передаем **this**.

LiveData получит из **Activity** его **Lifecycle** и по нему будет определять состояние Activity. Активным считается состояние STARTED или RESUMED. Т.е. если Activity видно на экране, то **LiveData** считает его активным и будет отправлять данные в его колбэк.

2) это непосредственно подписчик, т.е. колбэк, в который **LiveData** будет отправлять данные. В нем только один метод **onChanged**. В нашем примере туда будет приходить String.

Теперь, когда **DataController** поместит какой-либо **String** объект в **LiveData**, мы сразу получим этот объект в **Activity**, если **Activity** находится в состоянии **STARTED** или **RESUMED**.

Особенности:

Если Activity было не активно во время обновления данных в **LiveData**, то при возврате в активное состояние, его **observer** получит *последнее актуальное значение данных*.

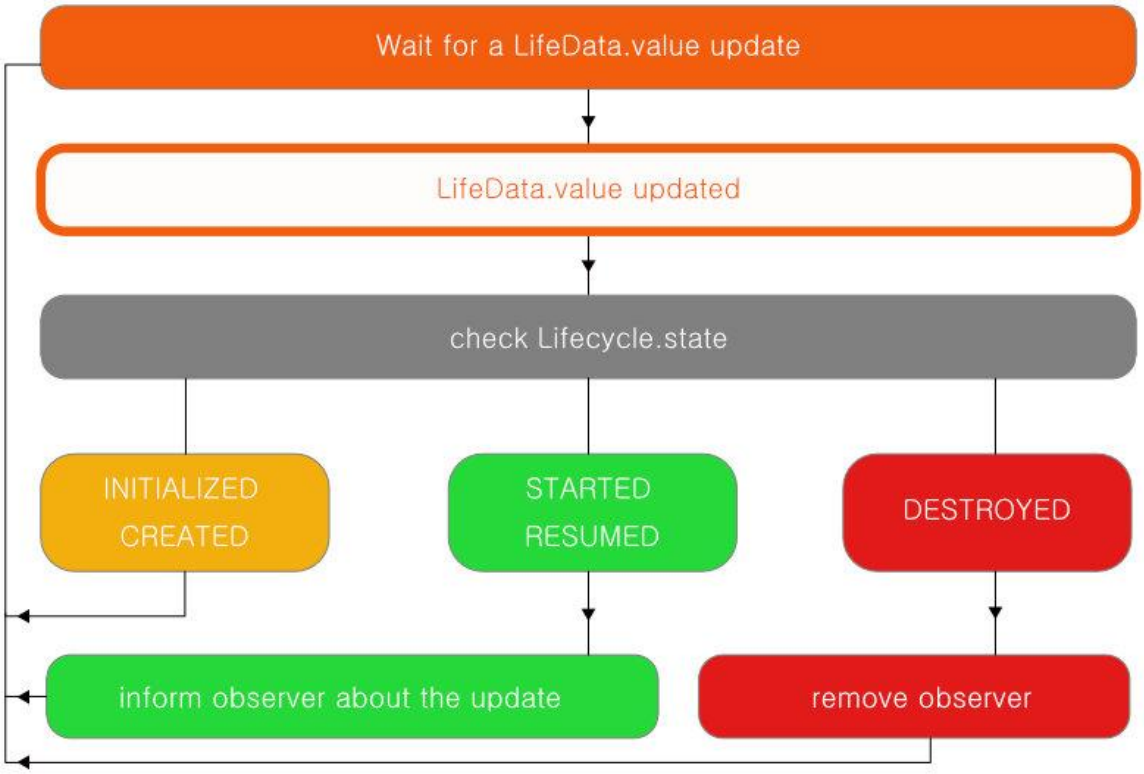
Если **Activity** будет закрыто, т.е. перейдет в статус DESTROYED, то **LiveData** *автоматически отпишет* от себя его **observer**.

Если Activity в состоянии DESTROYED попробует подписаться, то подписка не будет выполнена.

Если Activity уже подписывало свой observer, и попробует сделать это еще раз, то просто *ничего не произойдет*.

Вы всегда можете получить последнее значение **LiveData** с помощью его метода **getValue**.

Как видите, подписывать **Activity** на **LiveData** - это удобно. Поворот экрана и полное закрытие Activity - все это корректно и удобно обрабатывается автоматически без каких-либо усилий с нашей стороны.



Отправка данных в LiveData

В классе **DataController** переменная **LiveData** будет выглядеть так:

```
private MutableLiveData<String> liveData = new MutableLiveData<>();

LiveData<String> getData() {
    return liveData;
}
```

Наружу мы передаем **LiveData**, который позволит внешним объектам только получать данные. Но внутри **DataController** мы используем объект **MutableLiveData**, который позволяет помещать в него данные.

Чтобы поместить значение в **MutableLiveData**, используется метод **setValue**:

```
liveData.setValue("new value");
```

Этот метод обновит значение **LiveData**, и все его активные подписчики получат это обновление.

Метод **setValue** должен быть вызван из UI потока. Для обновления данных из других потоков используйте метод **postValue**. Он перенаправит вызов в UI поток. Соответственно, подписчики всегда будут получать значения в основном потоке.

Transformations

Вы можете поменять типы данных в **LiveData** с помощью **Transformations.map**.

Рассмотрим пример, в котором *LiveData<String>* будем превращать в *LiveData<Integer>:*

```
LiveData<String> liveData = ...;

LiveData<Integer> liveDataInt = Transformations.map(liveData,
        new Function<String, Integer>() {
            @Override
            public Integer apply(String input) {
                return Integer.parseInt(input);
            }
        });
```

В метод **map** передаем имеющийся **LiveData<String>** и функцию преобразования. В этой функции мы будем получать String данные из **LiveData<String>**, и от нас требуется преобразовать их в Integer. В данном случае просто парсим строку в число.

На выходе метода **map** получим **LiveData<Integer>**. Можно сказать, что он подписан на **LiveData<String>** и все получаемые **String** значения будет конвертировать в **Integer** и рассылать уже своим подписчикам.

Рассмотрим более сложный случай. У нас есть **LiveData<Long>**, нам необходимо из него получить **LiveData<User>**. Конвертация id в User выглядит так:

```
private LiveData<User> getUser(long id) {
    // ...
}
```

По id мы получаем **LiveData<User>** и на него надо будет подписываться, чтобы получить объект **User**.

В этом случае мы не можем использовать метод **map**, т.к. мы получим примерно такой результат:

```
LiveData<Long> liveDataId = ...;

LiveData<LiveData<User>> liveDataUser = Transformations.map(liveDataId,
        new Function<Long, LiveData<User>>() {
            @Override
            public LiveData<User> apply(Long id) {
        return getUser(id);
   }
});
```

На выходе будет объект **LiveData<LiveData<User>>**. Чтобы избежать этого, используем s**witchMap** вместо **map**.

```
LiveData<Long> liveDataId = ...;

LiveData<User> liveDataUser = Transformations.switchMap(liveDataId,
        new Function<Long, LiveData<User>>() {
            @Override
            public LiveData<User> apply(Long id) {
                return getUser(id);
            }
        });
```

switchMap уберет вложенность LiveData и мы получим LiveData<User>.

Свой LiveData

В некоторых ситуациях удобно создать свою обертку **LiveData**. Рассмотрим пример:

```
public class LocationLiveData extends LiveData<Location> {

    LocationService.LocationListener locationListener =
            new LocationService.LocationListener() {
                @Override
                public void onLocationChanged(Location location) {
                    setValue(location);
                }
            };

    @Override
    protected void onActive() {
        LocationService.addListener(locationListener);
    }

    @Override
    protected void onInactive() {
        LocationService.removeListener(locationListener);
    }

}
```

Класс **LocationLiveData** расширяет **LiveData<Location>.**

Внутри него есть некий **locationListener** - слушатель, который можно передать в **LocationService** и получать обновления текущего местоположения. При получении нового **Location** от **LocationService**, **locationListener** будет вызывать метод **setValue** и тем самым обновлять данные этого **LiveData**.

LocationService - это просто какой-то сервис, который предоставляет нам текущую локацию. Его реализация в данном примере не важна. Главное - это то, что мы подписываемся (**addListener**) на сервис, когда нам нужны данные, и отписываемся (**removeListener**), когда данные больше не нужны.

Обратите внимание, что мы переопределили методы **onActive** и **onInactive**. **onActive** будет вызван, когда у **LiveData** появится хотя бы один подписчик. А **onInactive** - когда не останется ни одного подписчика. Соответственно эти методы удобно использовать для подключения/отключения нашего слушателя к **LocationService**.

Получилась удобная обертка, которая при появлении подписчиков сама будет подписываться к **LocationService**, получать **Location** и передавать его своим подписчикам. А когда подписчиков не останется, то **LocationLiveData** отпишется от **LocationService**.

MediatorLiveData

MediatorLiveData дает возможность собирать данные из нескольких **LiveData** в один. Это удобно если есть несколько источников из которых вы хотите получать данные. Вы объединяете их в один и подписываетесь только на него.

Рассмотрим, как это делается, на простом примере.

```
MutableLiveData<String> liveData1 = new MutableLiveData<>();
MutableLiveData<String> liveData2 = new MutableLiveData<>();

MediatorLiveData<String> mediatorLiveData = new MediatorLiveData<>();


mediatorLiveData.addSource(liveData1, new Observer<String>() {
        @Override
        public void onChanged(@Nullable String s) {
            mediatorLiveData.setValue(s);
        }
    });


mediatorLiveData.addSource(liveData2, new Observer<String>() {
        @Override
        public void onChanged(@Nullable String s) {
            mediatorLiveData.setValue(s);

        }
    });
```

Метод **addSource** требует от нас два параметра.

Первый - это **LiveData** из которого **MediatorLiveData** собирается получать данные.

Второй параметр - это колбэк, который будет использован для подписки на **LiveData** из первого параметра. Обратите внимание, что в колбэке надо самим передавать в MediatorLiveData данные, получаемые из LiveData. Это делается методом **setValue**.

Таким образом **mediatorLiveData** будет получать данные из двух LiveData и постить их своим получателям.

Подпишемся на **mediatorLiveData**

```
mediatorLiveData.observe(this, new Observer<String>() {
        @Override
        public void onChanged(@Nullable String s) {
            log("onChanged " + s);
        }
    });
```

Сюда теперь должны приходить данные из *liveData1* и *liveData2*.

Немного усложним пример. Допустим, нам надо отписаться от *liveData2*, когда из него придет значение "finish".

Код подписки **mediatorLiveData** на *liveData1* и *liveData2* будет выглядеть так:

```
mediatorLiveData.addSource(liveData1, new Observer<String>() {
        @Override
        public void onChanged(@Nullable String s) {
            mediatorLiveData.setValue(s);
        }
    });


mediatorLiveData.addSource(liveData2, new Observer<String>() {
        @Override
        public void onChanged(@Nullable String s) {
            if ("finish".equalsIgnoreCase(s)) {
                mediatorLiveData.removeSource(liveData2);
                return;
            }
            mediatorLiveData.setValue(s);

        }
    });
```

В случае с *liveData1* ничего не меняется.

А вот при получении данных от *liveData2* мы смотрим, что за значение пришло. Если это значение "finish", то методом **removeSource** отписываем **mediatorLiveData** от *liveData2* и не передаем это значение дальше.

RxJava

Мы можем конвертировать **LiveData** в **Rx** и наоборот. Для этого есть инструмент **LiveDataReactiveStreams**.

Чтобы его использовать добавьте в dependencies:

```
implementation "androidx.lifecycle:lifecycle-
reactivestreams:$lifecycle_version"
```

чтобы получить **LiveData** из **Flowable** или **Observable**, используем метод **fromPublisher**:

```
Flowable<String> flowable = ... ;
LiveData<String> liveData = LiveDataReactiveStreams.fromPublisher(flowable);
```

LiveData будет подписан на **Flowable**, пока у него (у **LiveData**) есть подписчики.

LiveData не сможет обработать или получить onError от Flowable. Если в **Flowable** возникнет ошибка, то будет крэш.

Неважно в каком потоке работает **Flowable**, результат в LiveData всегда придет в UI потоке.

Чтобы получить **Flowable** или **Observable** из **LiveData** нужно выполнить два преобразования. Сначала используем метод **toPublisher**, чтобы получить **Publisher**. Затем полученный **Publisher** передаем в метод **Flowable.fromPublisher**:

```
LiveData<String> liveData = … ;
Flowable<String> flowable = Flowable.fromPublisher(
        LiveDataReactiveStreams.toPublisher(this, liveData));
```

Прочие методы LiveData

hasActiveObservers() - проверка наличия активных подписчиков

hasObservers() - проверка наличия любых подписчиков

observeForever (Observer<T> observer) - позволяет подписаться без учета Lifecycle. Т.е. этот подписчик будет всегда считаться активным.

removeObserver (Observer<T> observer) - позволяет отписать подписчика

removeObservers (LifecycleOwner owner) - позволяет отписать всех подписчиков, которые завязаны на Lifecycle от указанного LifecycleOwner.

ViewModel

https://developer.android.com/topic/libraries/architecture/viewmodel

ViewModel - класс, позволяющий Activity и фрагментам сохранять необходимые им объекты живыми при повороте экрана.

Создаем свой класс, наследующий **ViewModel**

```
public class MyViewModel extends ViewModel {

}
```

Пока оставим его пустым.

Чтобы добраться до него в Activity, нужен следующий код:

```
@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** передаем Activity. Тем самым мы получим доступ к провайдеру, который хранит все ViewModel для этого Activity.

Методом **get** 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.*

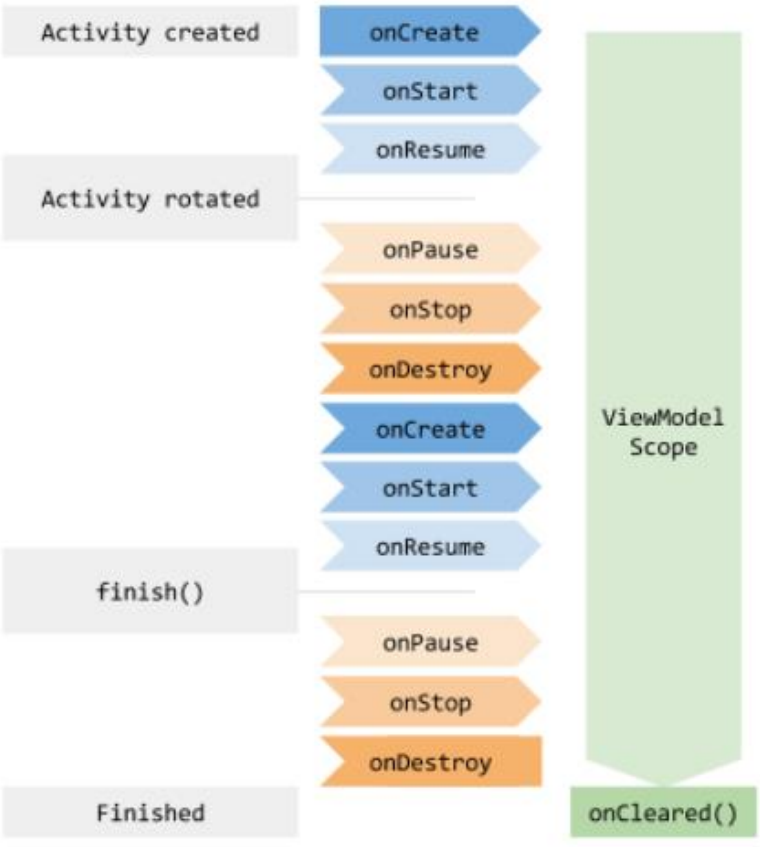
На картинке время жизни (оно же scope) модели это выглядит так.

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)
```

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



LiveData и ViewModel

LiveData очень удобно использовать с **ViewModel**.

Рассмотрим несложный пример асинхронной однократной загрузки каких-либо данных:

```
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:

```
@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.

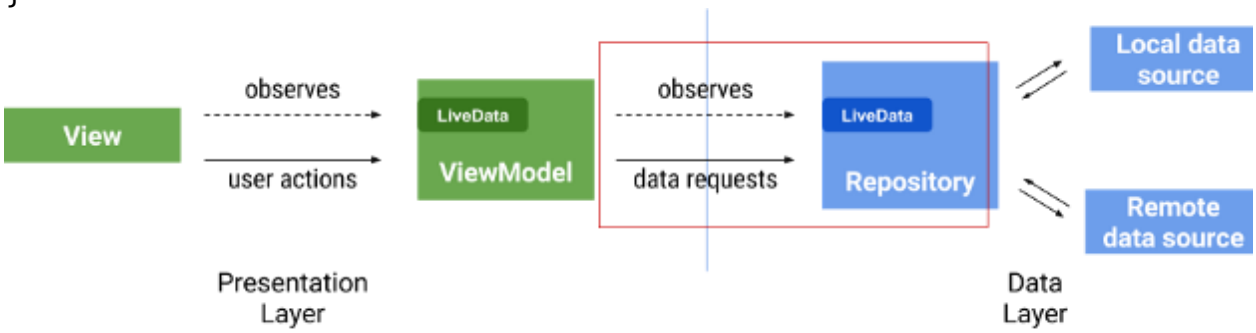
```
public class MyViewModel extends ViewModel {

    // ...

    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

```
public class MyViewModel extends ViewModel {

    // ...

    @Override
    protected void onCleared() {
        // clean up resources

    }
}
```

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**.

```
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**.

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

В методе **create** 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 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:

```
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

LiveData можно сравнить Flowable.

Но 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).

Библиотека Room

https://developer.android.com/topic/libraries/architecture/room

Библиотека **Room** предоставляет нам удобную обертку для работы с базой данных SQLite.

Чтобы использовать Room в приложении, добавьте артефакты компонентов архитектуры в файл **build.gradle** вашего приложения.

```
dependencies {
  def room_version = "2.2.5"

  implementation "androidx.room:room-runtime:$room_version"
  annotationProcessor "androidx.room:room-compiler:$room_version"

  // optional - RxJava support for Room
  implementation "androidx.room:room-rxjava2:$room_version"
}
```

Room имеет три основных компонента: **Entity**, **Dao** и **Database**. Рассмотрим их на небольшом примере, в котором будем создавать базу данных для хранения данных по сотрудникам (англ. - employee).

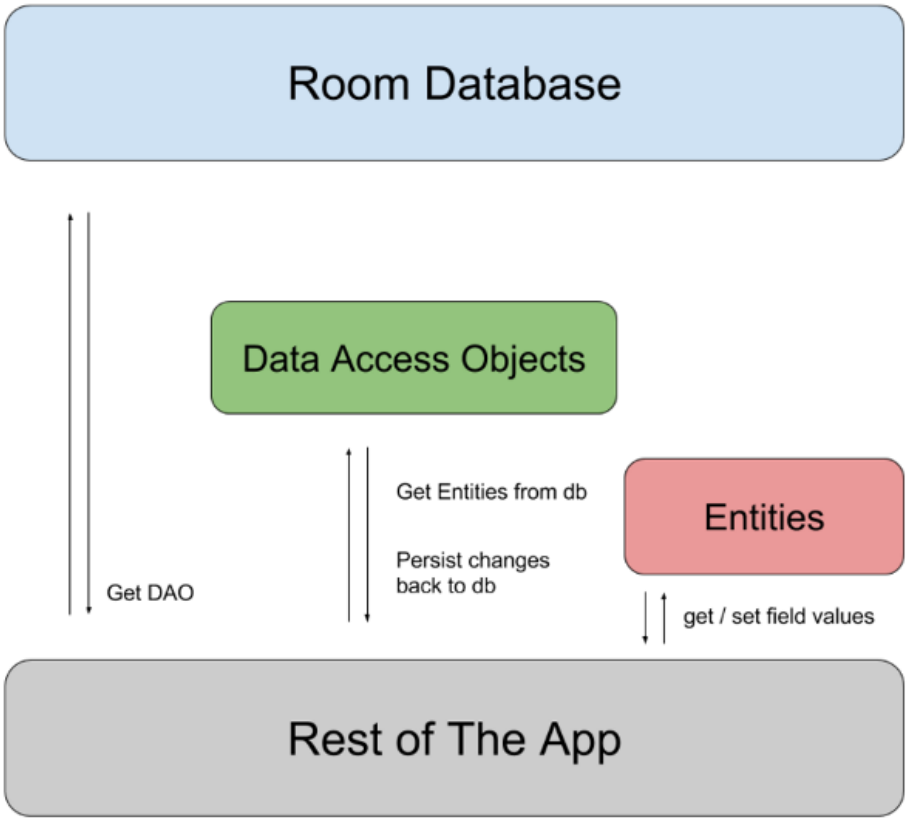
При работе с Room необходимо будет писать SQL запросы.

Entity: представляет таблицу в базе данных.

DAO: содержит методы, используемые для доступа к базе данных.

Приложение использует базу данных **Room**, чтобы получить объекты доступа к данным или **DAO**, связанные с этой базой данных. Затем приложение использует каждый **DAO** для получения сущностей из базы данных и сохранения любых изменений этих объектов обратно в базу данных. Наконец, приложение использует объект **Entity** для получения и установки значений, соответствующих столбцам таблицы в базе данных.

Эта взаимосвязь между различными компонентами Room представлена на рисунке:



Entity

Аннотацией **Entity** нам необходимо пометить объект, который мы хотим хранить в базе данных. Для этого создаем класс Employee, который будет представлять собой данные сотрудника:

```
@Entity
public class Employee {

    @PrimaryKey
    public long id;

    public String name;

    public int salary;
}
```

https://developer.android.com/training/data-storage/room/defining-data
https://developer.android.com/reference/androidx/room/package-summary?hl=en

Класс помечается аннотацией **Entity**. Объекты класса **Employee** будут использоваться при работе с базой данных. Например, мы будем получать их от базы при запросах данных и отправлять их в базу при вставке данных.

Этот же класс **Employee** будет использован для создания таблицы в базе. В качестве имени таблицы будет использовано имя класса. А поля таблицы будут созданы в соответствии с полями класса.

Аннотацией **PrimaryKey** мы помечаем поле, которое будет ключом в таблице.

Dao

В объекте **Dao** мы будем описывать методы для работы с базой данных. Нам нужны будут методы для получения списка сотрудников и для добавления/изменения/удаления сотрудников.

Описываем их в интерфейсе с аннотацией **Dao**.

```
@Dao
public interface EmployeeDao {

    @Query("SELECT * FROM employee")
    List<Employee> getAll();

    @Query("SELECT * FROM employee WHERE id = :id")
    Employee getById(long id);

    @Insert
    void insert(Employee employee);

    @Update
    void update(Employee employee);

    @Delete
    void delete(Employee employee);

}
```

https://developer.android.com/training/data-storage/room/accessing-data

Методы **getAll** и **getById** позволяют получить полный список сотрудников или конкретного сотрудника по id. В аннотации **Query** нам необходимо прописать соответствующие SQL-запросы, которые будут использованы для получения данных.

Обратите внимание, что в качестве имени таблицы используем employee. Имя таблицы равно имени **Entity** класса, т.е. **Employee**, но в SQLite не важен регистр в именах таблиц, поэтому можем писать employee.

Для вставки/обновления/удаления используются методы **insert/update/delete** с соответствующими аннотациями. Тут никакие запросы указывать не нужно. Названия методов могут быть любыми. Главное - аннотации.

Database

https://developer.android.com/training/data-storage/room/prepopulate

Аннотацией **Database** помечаем основной класс по работе с базой данных. Этот класс должен быть абстрактным и наследовать **RoomDatabase**.

```
@Database(entities = {Employee.class}, version = 1)

public abstract class AppDatabase extends RoomDatabase {
    public abstract EmployeeDao employeeDao();
}
```

В параметрах аннотации **Database** указываем, какие **Entity** будут использоваться, и версию базы. Для каждого **Entity** класса из списка entities будет создана таблица.

В Database классе необходимо описать абстрактные методы для получения Dao объектов.

Практика

Все необходимые для работы объекты созданы. Давайте посмотрим, как использовать их для работы с базой данных.

Database объект - это стартовая точка. Его создание выглядит так:

```
AppDatabase db =  Room.databaseBuilder(getApplicationContext(),
        AppDatabase.class, "database").build();
```

Используем Application Context, а также указываем AppDatabase класс и имя файла для базы.

Учитывайте, что при вызове этого кода Room каждый раз будет создавать новый экземпляр **AppDatabase**. Эти экземпляры очень тяжелые и рекомендуется использовать один экземпляр для всех операций. Поэтому необходимо позаботиться о синглтоне для этого объекта. Это можно сделать с помощью **Dagger**, например.

Если вы не используете Dagger (или другой DI механизм), то можно использовать Application класс для создания и хранения AppDatabase:

```
public class App extends Application {

    public static App instance;

    private AppDatabase database;

    @Override
    public void onCreate() {
        super.onCreate();
```

```
        instance = this;
        database = Room.databaseBuilder(this, AppDatabase.class,
"database")
                .build();
    }

    public static App getInstance() {
        return instance;
    }

    public AppDatabase getDatabase() {
        return database;
    }
}
```

В коде получение базы будет выглядеть так:

```
AppDatabase db = App.getInstance().getDatabase();
```

Из Database объекта получаем Dao.

```
EmployeeDao employeeDao = db.employeeDao();
```

Теперь мы можем работать с Employee объектами. Но эти операции должны выполняться не в UI потоке. Иначе мы получим Exception.

Добавление нового сотрудника в базу будет выглядеть так:

```
    Employee employee = new Employee();
    employee.id = 1;
    employee.name = "John Smith";
    employee.salary = 10000;

employeeDao.insert(employee);
```

Метод getAll вернет нам всех сотрудников в List<Employee>

```
List<Employee> employees = employeeDao.getAll();
```

Получение сотрудника по id:

```
Employee employee = employeeDao.getById(1);
```

Обновление данных по сотруднику.

```
employee.salary = 20000;
employeeDao.update(employee);
```

Room будет искать в таблице запись по ключевому полю, т.е. по id. Если в объекте employee не заполнено поле id, то по умолчанию в нашем примере оно будет равно нулю и Room просто не найдет такого сотрудника (если, конечно, у вас нет записи с id = 0).

Удаление сотрудника

```
employeeDao.delete(employee);
```

Аналогично обновлению, Room будет искать запись по ключевому полю, т.е. по id

Давайте для примера добавим еще один тип объекта - Car.

Описываем Entity объект

```
@Entity
public class Car {

    @PrimaryKey
    public long id;

    public String model;

    public int year;

}
```

Теперь Dao для работы с Car объектом

```
@Dao
public interface CarDao {

    @Query("SELECT * FROM car")
    List<Car> getAll();

    @Insert
    void insert(Car car);

    @Delete
    void delete(Car car);

}
```

Будем считать, что нам надо только читать все записи, добавлять новые и удалять старые.

В Database необходимо добавить Car в список entities и новый метод для получения CarDao

```
@Database(entities = {Employee.class, Car.class}, version = 2)

public abstract class AppDatabase extends RoomDatabase {
    public abstract EmployeeDao employeeDao();
    public abstract CarDao carDao();
}
```

Т.к. мы добавили новую таблицу, изменилась структура базы данных. И нам необходимо поднять версию базы данных до 2.

UI поток

Операции по работе с базой данных должны выполняться не в UI потоке.

В случае с Query операциями мы можем сделать их асинхронными используя LiveData или RxJava.

В случае insert/update/delete вы можете обернуть эти методы в асинхронный RxJava.

Также, вы можете использовать allowMainThreadQueries в билдере создания AppDatabase

```
AppDatabase db =  Room.databaseBuilder(getApplicationContext(),
        AppDatabase.class, "database")
        .allowMainThreadQueries()
        .build();
```

В этом случае вы не будете получать Exception при работе в UI потоке. Но должны понимать, что это плохая практика, и может добавить ощутимыого замедления вашему приложению.

Переход на Room

Если вы надумали с SQLite мигрировать на Room, то вот пара полезных ссылок по этой теме:

https://medium.com/google-developers/incrementally-migrate-from-sqlite-to-room-66c2f655b377

https://medium.com/@price.yvonne.86/quick-and-easy-migration-to-room-d40dbb142b51

https://devcolibri.com/5-common-mistakes-when-using-architecture-components/