МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Лабораторная работа № 2/4ч.

Разработка мобильных компонент анализа безопасности информационно-аналитических систем

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень: бакалавриат
(бакалавриат, магистратура, специалитет)

Форма обучения: очная
(очная, очно-заочная, заочная)

Направление(-я) подготовки: 10.05.04 Информационно-аналитические системы безопасности
(код(-ы) и наименование(-я))

Институт: комплексной безопасности и специального приборостроения ИКБСП
(полное и краткое наименование)

Кафедра: КБ-4 «Прикладные информационные технологии»
(полное и краткое наименование кафедры, реализующей дисциплину (модуль))

Используются в данной редакции с учебного года 2021/22
(учебный год цифрами)

Проверено и согласовано «____» _____20___г. _____ _____
(подпись директора Института/Филиала с расшифровкой)

Москва 2021 г.

ОГЛАВЛЕНИЕ

1 ANDROID ARCHITECTURE COMPONENTS

Android Architecture Components — это набор библиотек от Google, предназначенные для проектирования, тестирования и сопровождения приложений. В новом руководстве по архитектуре Android определены некоторые ключевые принципы, которые должны соответствовать хорошему Android-приложению, а также оно предлагает безопасный путь для разработчика по созданию хорошего приложения. Согласно руководству, хорошее приложение для Android должно обеспечить четкое разделение обязанностей и управлять пользовательским интерфейсом раздельно от модели. Любой код, который не обрабатывает взаимодействие с пользовательским интерфейсом или операционной системой, не должен находиться в Activity или Fragment, поскольку сохранение их как можно более чистыми позволит вам избежать многих проблем, связанных с жизненным циклом приложения. В конце концов, система может уничтожить действия или фрагменты в любое время. Кроме того, данные должны обрабатываться с помощью моделей, которые изолированы от пользовательского интерфейса и, следовательно, от проблем жизненного цикла.

Чтобы понять, что предлагает команда Android, мы должны знать все элементы компонентов архитектуры, так как именно они будут делать все тяжелую работу для нас. Есть четыре основных компонента, каждый из которых имеет определенную роль: Room, ViewModel, LiveData и Lifecycle. У всех этих частей есть свои обязанности, и они работают вместе, чтобы создать прочную архитектуру. Давайте рассмотрим упрощенную схему предлагаемой архитектуры, чтобы лучше понять ее.

1. Activity и Fragment Lifecycle - предоставляют несколько механизмов, сочетание которых, позволяет удобно обрабатывать повороты экрана.

– Lifecycle - отслеживает текущий статус Activity и имеет возможность уведомлять об этом своих подписчиков;

– LiveData - получает и хранит данные, может отправлять их своим подписчикам;

– ViewModel - сохраняет необходимые объекты при повороте экрана.

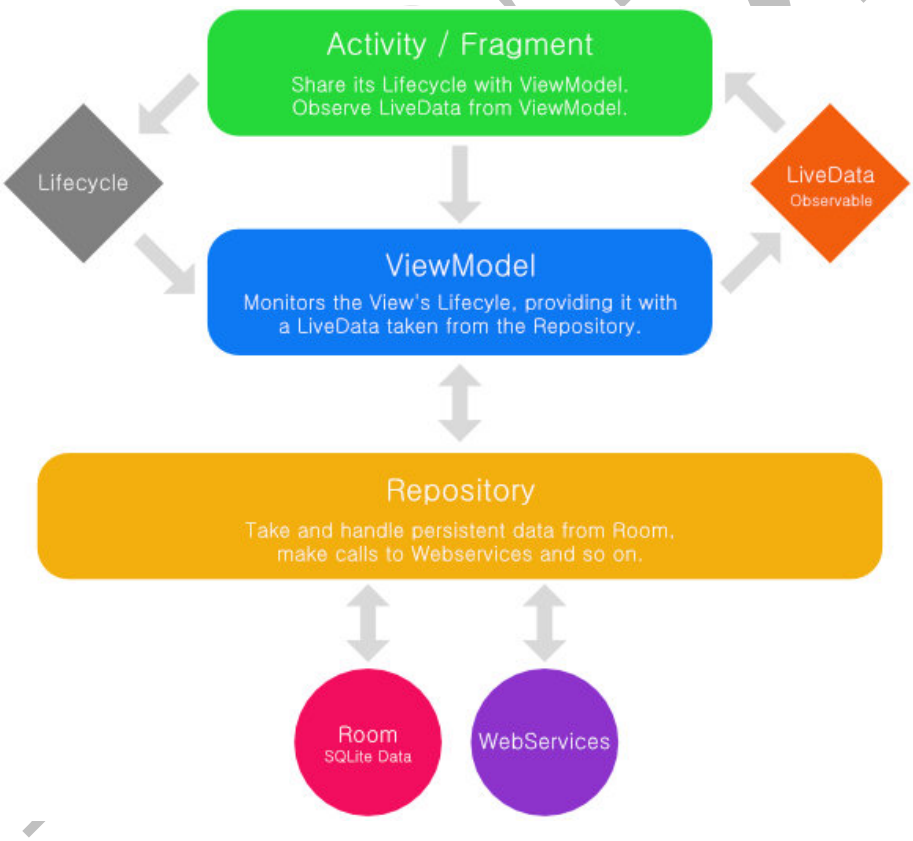
2. Room - удобная обертка для работы с базой данных

3. Paging Library - библиотека для постраничной загрузки данных из базы данных, с сервера или любого другого источника.

4. Data Binding - связывание View’s с объектами: отображение данных, запись изменений в данные полученные с View, объявление переменных (только объекты) в xml-разметке и взаимодействие с ними непосредственно в xml.

5. Navigation Architecture Component - новый компонент для навигации по экранам приложения

6. WorkManager - позволяет запускать фоновые задачи последовательно или параллельно, передавать в них данные, получать из них результат, отслеживать статус выполнения и запускать только при соблюдении заданных условий.



1.1 Activity и Fragment Lifecycle. Lifecycle

Компонент Lifecycle – предназначен для работы с жизненным циклом. Выделены основные понятия такие как:

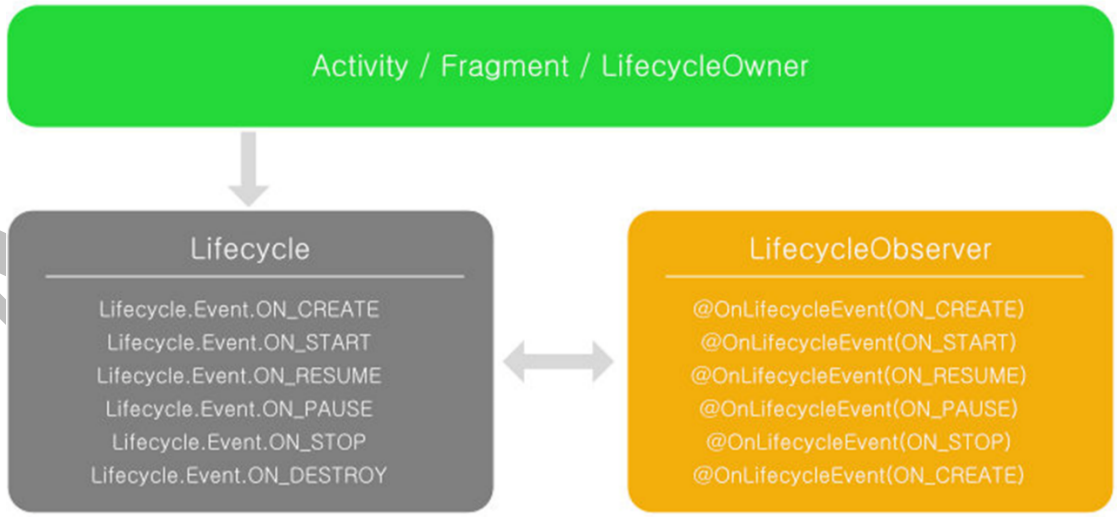
- LifecycleOwner – это интерфейс с одним методом getLifecycle(), который возвращает состояние жизненного цикла. Являет собой абстракцию владельца

жизненного цикла (Activity, Fragment). Для упрощения добавлены классы LifecycleActivity и LifecycleFragment.

- LifecycleObserver – интерфейс, обозначает слушателя жизненного цикла owner-а. Не имеет методов, завязан на OnLifecycleEvent, который в свою очередь разрешает отслеживать жизненный цикл. Для описания состояния есть два enum. Первый Events — который обозначает изменение цикла и второй State — который описывает текущее состояние.

- Events — повторяет стадии жизненного цикла и состоит из ON_CREATE, ON_RESUME, ON_START, ON_PAUSE, ON_STOP, ON_DESTROY, а также ON_ANY который информирует про изменения состояния без привязки к конкретному этапу.

State — состоит из следующих констант: INITIALIZED, CREATED, STARTED, RESUMED, DESTROYED. Для получения состояния используется метод getCurrentState() из Lifecycle. Также в Enum State реализован метод itAtLeast(State), который отвечает на вопрос является State выше или равным от переданного как параметр.



Довольно частая ситуация, когда в приложении работает ряд процессов, которые зависят от этапа жизненного цикла. Будь-то воспроизведение медиа, локация, связь с сервисом и т.д. Таким образом приходится вручную отслеживать состояние и уведомлять о нём процесс. ИТОГ: захламление основного класса

(Activity или Fragment) и снижение модульности, ведь требуется поддержка передачи состояния. С помощью же данного компонента возможно переложить всю ответственность на компонент и все что для этого нужно это объявить интересующий наш класс как observer и передать ему в onCreate() методе ссылку на owner.

1.2 Задание

File> New> New Module> Phone & Tablet Module> Empty Activity. Название приложения lifecycle

Activity и фрагменты в Support Library, начиная с версии 26.1.0 реализуют интерфейс LifecycleOwner. Данный интерфейс добавляет метод getLifecycle. Т.е. версия, указанная в build.gradle файле модуля должна быть не ниже 26.1.0:

```
android {
    compileSdkVersion 29
    buildToolsVersion "29.0.2"
...
}
```

У Activity есть метод getLifecycle, который возвращает объект Lifecycle. Lifecycle — класс, который хранит информацию про состояние жизненного цикла и разрешает другим объектам отслеживать его с помощью реализации LifecycleObserver. Состоит из методов: addObserver(LifecycleObserver), removeObserver(LifecycleObserver) и getCurrentState(). Как понятно из названий для добавления подписчика, удаления и соответственно получения текущего состояния.

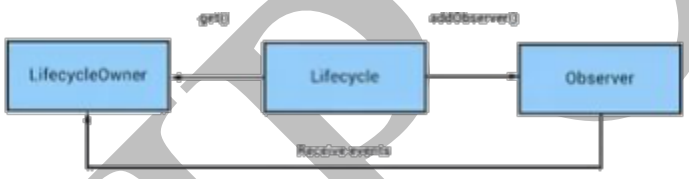
Для описания состояния есть два enum:

- **Events** — повторяет стадии жизненного цикла и состоит из ON_CREATE, ON_RESUME, ON_START, ON_PAUSE, ON_STOP, ON_DESTROY, а также ON_ANY который информирует про изменения состояния без привязки к конкретному этапу. Отслеживание изменений цикла происходит с помощью пометки метода в обсервере аннотацией OnLifecycleEvent, которому как параметр передается интересуещее нас событие.

```
@OnLifecycleEvent(Lifecycle.Event.ON_ANY)
void stateUpdated() {
    //будет вызваться при каждом изменении состояния жизненого цикла у activity.
}
```

- **State** — состоит из следующих констант: INITIALIZED, CREATED, STARTED, RESUMED, DESTROYED. Для получения состояния используется метод getCurrentState() из Lifecycle. Также в Enum State реадизован метод itAtLeast(State), который отвечает на вопрос являтся State выше или равным от переданого как параметр.

Для работы с компонентом Lifecycle, требуется определить **owner**, то есть владельца жизненного цикла и **observer**, того кто на него будет подписан. У owner может быть любое количество подписчиков, также стоит отметить что observer будет проинформирован про изменение состояния, еще до того как у owner будет вызван метод super() на соответствующий метод жизненного цикла.



Owner должен реализовывать интерфейс LifecycleOwner, который содержит один метод getLifecycle(), который возвращает екземпляр класса холдера Lifecycle.

Observer должен реализовать интерфейс маркер LifecycleObserver. Требуется создать класс Server, который будет устанавливать соединение с сервером:

```
public class Server implements LifecycleObserver {
    private String TAG = "lifecycle";

    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    public void connect() {
        Log.d(TAG,"connect to web-server");
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
    public void disconnect() {
        Log.d(TAG,"disconnect");
    }
}
```

Следует обратить внимание, что интерфейс LifecycleObserver пустой. В нем нет методов типа onStart, onStop и т.п. Таким образом требуется просто отметить в

классе Server его же собственные методы аннотацией OnLifecycleEvent и указать, при каком lifecycle-событии метод должен быть вызван.

В нашем случае требуется указать, что метод connect() должен вызываться в момент onStart, а метод disconnect - в момент onStop.

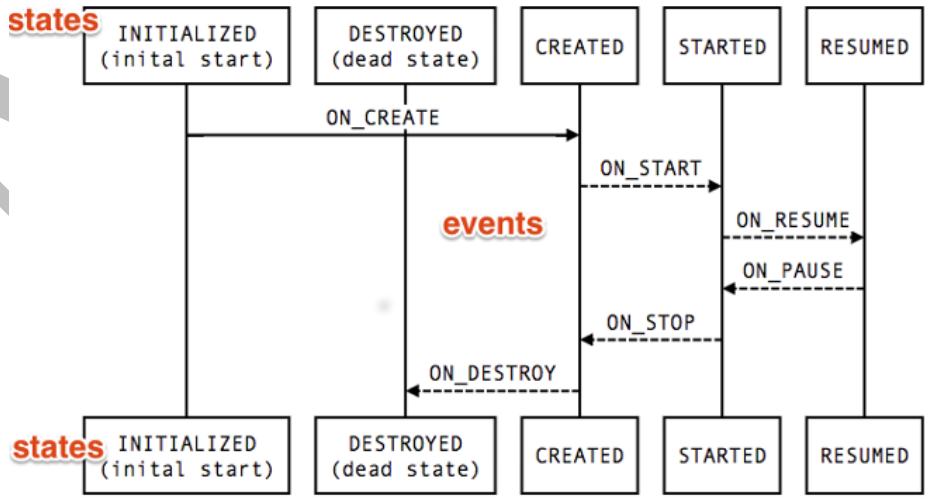
Осталось связать экземпляр MyServer и Lifecycle:

```
public class MainActivity extends AppCompatActivity {
    private Server server;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        server = new Server();
        getLifecycle().addObserver(server);
    }
}
```

В Activity методом getLifecycle возвращается Lifecycle, и методом addObserver производится подпись myServer на события. 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 минимум кода - только подписать server на Lifecycle. Все остальное решает сам Server.

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



На данной схеме отображены состояния и события. Они связаны следующим

образом - при переходе между состояниями происходят события.

Эти события указываются в аннотациях OnLifecycleEvent к методам объекта MyServer.

Запустите программу, попробуйте повернуть экран, исследуйте логи. Добавьте дополнительные методы, чтобы отслеживать все состояния Activity.

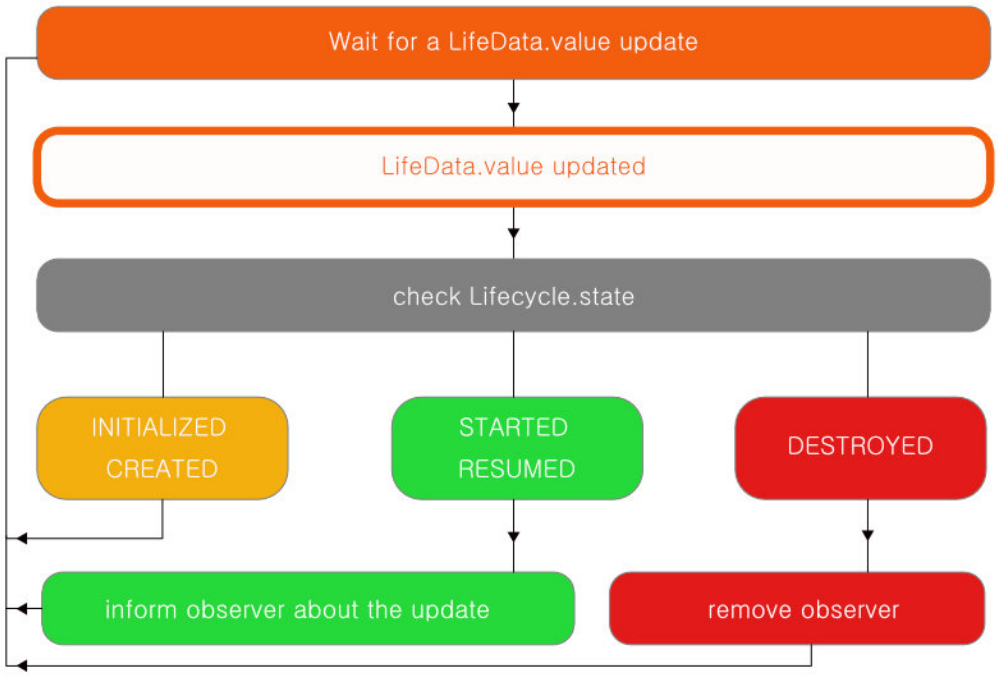
1.3 LiveData

Компонент LiveData – хранилище данных, работающее по принципу паттерна Observer (наблюдатель). Возможны следующие операции:

- поместить какой-либо объект для хранения;
- 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.

Например, каналы в мессенджерах. Автор пишет пост и отправляет его в канал, а все подписчики получают этот пост.

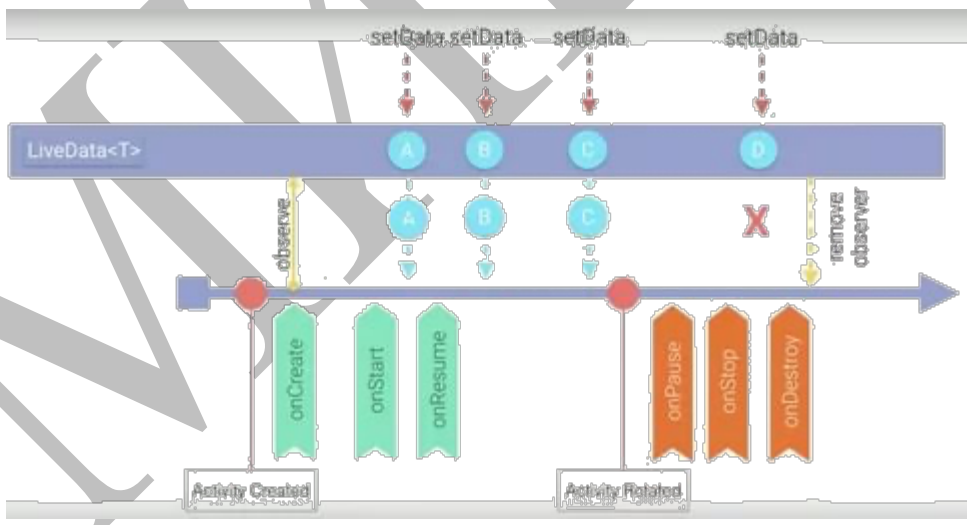


LiveData умеет определять активен подписчик или нет, и отправлять данные будет только активным подписчикам. Предполагается, что подписчиками LiveData будут Activity и фрагменты. А их состояние активности будет определяться с помощью их Lifecycle объекта.

Схема поведения LiveData представлена ниже:

- если Activity было не активно во время обновления данных в LiveData, то при возврате в активное состояние, его observer получит последнее актуальное значение данных;
- в момент подписки, observer получит последнее актуальное значение из LiveData;
- 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;
- всегда возможно получить последнее значение LiveData с помощью его метода getValue.

Поворот экрана и полное закрытие Activity - все это корректно и удобно обрабатывается автоматически без каких-либо усилий со стороны разработчика.



Сам компонент состоит из классов: LiveData, MutableLiveData, MediatorLiveData, LiveDataReactiveStreams, Transformations и интерфейса: Observer.

1.4 Задание

File> New> New Module> Phone & Tablet Module> Empty Activity. Название приложения livedata

Требуется создать класс TimeController на основе ViewModel с использованием LiveData. Данный класс предназначен для хранения времени.

В методе getData() возвращается объект LiveData, который позволит внешним объектам только получать данные. Но внутри LiveDataTimeController используется объект MutableLiveData, который позволяет помещать в него данные.

Чтобы поместить значение в MutableLiveData, используется метод setValue. Этот метод обновит значение LiveData, и все его активные подписчики получат это обновление.

```
class TimeLiveData {
    private static MutableLiveData<Long> data = new MutableLiveData<Long>();
    //sets latest time to LiveData
    static LiveData<Long> getTime(){
        data.setValue(new Date().getTime());
        return data;
    }
    static void setTime(){
        data.setValue(new Date().getTime());
    }
}
```

Код в Activity выглядит следующим образом:

```
public class MainActivity extends AppCompatActivity implements Observer<Long> {
    private TextView networkNameTextView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        networkNameTextView = findViewById(R.id.textView);
        TimeLiveData.getTime().observe(this, this);                    1
        Handler handler = new Handler();
        handler.postDelayed(new Runnable() {
            @Override
            public void run() {
                TimeLiveData.setTime();                                2
            }
        }, 5000);
    }

    @Override
    public void onChanged(@Nullable Long s) {
        Log.d(MainActivity.class.getSimpleName(), s + "");             3
        networkNameTextView.setText("" + s);
    }
}
```

Производится получение LiveData из LiveDataTimeController, и методом observe подписка на оповещение (1). В метод observe передаётся два параметра:

- LifecycleOwner - это интерфейс с методом getLifecycle. Activity и фрагменты в Support Library, начиная с версии 26.1.0 реализуют этот интерфейс, поэтому передаётся **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.
- Второй параметр - это непосредственно подписчик, т.е. колбэк, в который LiveData будет отправлять данные. В нем только один метод onChanged. В данном примере возвращается значение типа String.

Теперь, когда DataController поместит какой-либо String объект в LiveData (2 в данном случае activity само устанавливает данное значение), сразу возвращается данный объект в Activity в метод onChanged(3), если Activity находится в состоянии STARTED или RESUMED. Метод setTime должен быть вызван из UI потока. Для обновления данных из других потоков используется метод postValue. Он

перенаправит вызов в UI поток. Соответственно, подписчики всегда будут получать значения в основном потоке.

Если Activity было не активно во время обновления данных в LiveData, то при возврате в активное состояние, его observer получит последнее актуальное значение данных. В момент подписки, observer получит последнее актуальное значение из LiveData.
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 ничего не произойдет.
Всегда возможно получить последнее значение LiveData с помощью его метода getValue.

Подписывать Activity на LiveData достаточно удобно и просто. Поворот экрана и полное закрытие Activity - все это корректно и удобно обрабатывается автоматически без каких-либо усилий со стороны разработчика.

1.5 Transformations

Возможно изменять типы данных в LiveData с помощью Transformations.map.
Рассмотрим пример, в котором LiveData<Long> преобразуется в LiveData<String> из прошлого примера:

```
class TimeLiveData {
    private static MutableLiveData<Long> data = new MutableLiveData<Long>();

    //sets latest time to TimeLiveData
    static LiveData<Long> getTime(){
        data.setValue(new Date().getTime());
        return data;
    }
    static void setTime(){
        data.setValue(new Date().getTime());
    }
    // преобразование long в дату
    private static LiveData getStringTime = Transformations.map(data, new
Function<Long, String>() {
        @Override
        public String apply(Long input) {
            Calendar calendar = Calendar.getInstance();
            SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            return df.format(calendar.getTime());
        }
    });
    static LiveData<String> getDate(){
        return getStringTime;
    }
}
```

В метод map передаётся имеющийся LiveData<Long> и функция преобразования. В данной функции производится преобразование Long в понятную для человека форму отображения текущей даты и времени.

На выходе метода map возвращается LiveData<String>. Теперь требуется изменить в MainActivity тип возвращаемого значения в Observer:

```
public class MainActivity extends AppCompatActivity implements Observer<String> {
    private TextView networkNameTextView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        networkNameTextView = findViewById(R.id.textView);

        TimeLiveData.getDate().observe(this, this);
        Handler handler = new Handler();
        handler.postDelayed(new Runnable() {
            @Override
            public void run() {
                TimeLiveData.setTime();
            }
        }, 5000);
    }
    @Override
    public void onChanged(@Nullable String s) {
        Log.d(MainActivity.class.getSimpleName(), s + "");
        networkNameTextView.setText(""+ s);
    }
}
```

1.6 Задание

File> New> New Module> Phone & Tablet Module> Empty Activity. Название приложения networkstate. Требуется отслеживать состояние сети с помощью livedata.

Так как работа будет осуществляться с сетевой часть, в манифесте требуется установить соответствующие разрешения:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

Класс LiveData содержит дополнительные методы позволяющие:

- onActive() – этот метод вызывается когда у экземпляра есть активный(-ые) наблюдатели. В нем требуется инициировать интересующий сервис или операцию.
- onInactive() – вызывается когда у LiveData нет активных слушателей. Соответственно нужно остановить сервис или операцию.
- setValue() – вызывается если изменились данные и LiveData информирует

об этом слушателей.

- postValue(T) - передать значение из другого потока.

Далее создаётся класс, отвечающий за получение текущего состояния сети:

```
public class NetworkLiveData extends LiveData<String> {
    private Context context;
    private BroadcastReceiver broadcastReceiver;
    private static NetworkLiveData instance;
    static NetworkLiveData getInstance(Context context) {
        if (instance == null) {
            instance = new NetworkLiveData(context.getApplicationContext());
        }
        return instance;
    }

    private NetworkLiveData(Context context) {
        if (instance != null) {
            throw new RuntimeException("Use getInstance() method to get the single instance of this
class.");
        }
        this.context = context;
    }
    // Отслеживание изменения состояния сети осуществляется через BroadcastReceiver. При
изменении состояния генерируется
    // сообщение для всех слушателей в системе.
    private void prepareReceiver(Context context) {
        IntentFilter filter = new IntentFilter();
        filter.addAction("android.net.conn.CONNECTIVITY_CHANGE");
        broadcastReceiver = new BroadcastReceiver() {
            @Override
            public void onReceive(Context context, Intent intent) {
                ConnectivityManager cm =
                        (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY_SERVICE);
                NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
                if (activeNetwork != null) {
                    boolean isConnected = activeNetwork.isConnectedOrConnecting();
                    setValue(Boolean.toString(isConnected));
                }
                else
                    setValue("false");
            }
        };
        context.registerReceiver(broadcastReceiver, filter);
    }
    @Override
    protected void onActive() {
        prepareReceiver(context);
    }

    @Override
    protected void onInactive() {
        context.unregisterReceiver(broadcastReceiver);
        broadcastReceiver = null;
    }
}
```

Логика класса заключается в следующем, если кто-то подписывается, происходит инициализация BroadcastReceiver, который будет уведомлять

слушателей об изменении сети, после того как отписывается последний подписчик отслеживание изменения сети прекращается.

Для того чтобы добавить подписчика есть два метода: observe(LifecycleOwner, Observer<T>) — для добавления подписчика с учетом жизненного цикла и observeForever(Observer<T>) — без учета. Уведомления об изменении данных приходят с помощью реализации интерфейса Observer, который имеет один метод onChanged(T).

Получение данных производится в Activity:

```
public class MainActivity extends AppCompatActivity {
    private TextView textView;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        textView = findViewById(R.id.textView);
        LiveData<String> networkLiveData = NetworkLiveData.getInstance(this);
        networkLiveData.observe(this, new Observer<String>() {
            @Override
            public void onChanged(@Nullable String value) {
                textView.setText(value);
            }
        });
    }
}
```

Запустите приложение! Включите/отключите авиарежим.

1.7 ViewModel

Компонент ViewModel — предназначен для хранения и управления данными, связанными с представлением, решает проблемы, связанные с пересозданием активити во время таких операций, как переворот экрана и т.д. Данный компонент не является заменой onSaveInstanceState, поскольку, после того как система уничтожит активити, к примеру, когда происходит переход в другое приложение, ViewModel будет также уничтожена и не сохранит свое состояние. В целом же, компонент ViewModel является синглтон с колекцией экземпляров классов ViewModel, который гарантирует, что не будет уничтожен пока есть активный экземпляр активити и произойдёт освобождение ресурсов после ухода с нее. Стоит также отметить, что нельзя привязать любое количество ViewModel к

Activity(Fragment). Компонент состоит из таких классов: *ViewModel, AndroidViewModel, ViewModelProvider, ViewModelProviders, ViewModelStore, ViewModelStores*. Разработчик взаимодействует только с ViewModel, AndroidViewModel и для получения инстанса с ViewModelProviders.

Класс ViewModel, представляет абстрактный класс, без абстрактных методов и с одним protected методом onCleared(). Для реализации собственного ViewModel, необходимо унаследовать свой класс от ViewModel с конструктором без параметров. Если же требуется очистить ресурсы, то необходимо переопределить метод onCleared(), который будет вызван когда ViewModel долго не доступна и должна быть уничтожена. Стоит еще добавить, что во избежание утечки памяти, не нужно ссылаться напрямую на View или Context Activity из ViewModel. В целом, ViewModel должна быть абсолютно изолированная от представления данных. С помощью LiveData происходит уведомление уровнем представления (Activity/Fragment) об изменениях в наших данных.

1.8 Задание:

File> New> New Module> Phone & Tablet Module> Empty Activity. Название приложения ViewModel. Требуется, чтобы ProgressBar отображался в течении 10 секунд, если был произведён поворот экрана таймер не должен перезапускаться.

Добавить в разметку activity_main.xml элемент progressBar.

Все изменяемые данные должны сохранять с помощью LiveData, если же необходимо, к примеру, показать и скрыть ProgressBar, возможно создать MutableLiveData и хранить логику показать\скрыть в компоненте ViewModel. Далее создаётся класс ProgressViewModel, отвечающий за логику отображения ProgressBar.

```
class ProgressViewModel extends ViewModel {
    private static MutableLiveData<Boolean> isShowProgress = new MutableLiveData<>();

    // Показать прогресс в течении 10 сек.
    void showProgress() {
        isShowProgress.postValue(true);
        new Handler().postDelayed(new Runnable() {
            @Override
            public void run() {
                isShowProgress.postValue(false);
            }
        }, 10000);
    }
    // Возвращает состояние прогресс?
    MutableLiveData<Boolean> getProgressState() {
        return isShowProgress;
    }
}
```

Для получения ссылки на экземпляр ProgressViewModel требуется воспользоваться ViewModelProviders:

```
public class MainActivity extends AppCompatActivity {
    private ProgressBar progressBar;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        progressBar = findViewById(R.id.progressBar);
        // получение доступа к создайнной ViewModel
        ProgressViewModel viewModel =
ViewModelProviders.of(this).get(ProgressViewModel.class);
        viewModel.getProgressState().observe(this, new Observer<Boolean>() {
            @Override
            public void onChanged(@Nullable Boolean isVisibleProgressBar) {
                if (isVisibleProgressBar) {
                    progressBar.setVisibility(View.VISIBLE);
                } else {
                    progressBar.setVisibility(View.GONE);
                }
            }
        });
        viewModel.showProgress();
    }
}
```

Запустите приложение! При запуске отображается ProgressBar и начинается отсчёт 10 сек. При повороте устройства значение, указывающее на отображение ProgressBar сохраняется в MutableLiveData. ViewModel позволяет сообщить о состоянии элементов Activity. Описание алгоритма приведено далее.

1.8.1 Компоненты ViewModel

Класс **AndroidViewModel**, являет собой расширение ViewModel, с единственным отличием — в конструкторе должен быть один параметр Application. Является довольно полезным расширением в случаях, когда требуется использовать Location Service или другой компонент, требующий Application Context. Основным отличием является то, что наследуется ViewModel от ApplicationViewModel. В Activity/Fragment процедура инициализации производится аналогично с ViewModel.

Класс **ViewModelProviders**, являет собой четыре метода утилиты, которые, называются of и возвращают ViewModelProvider. Адаптированные для работы с Activity и Fragment, а также, с возможностью подставить свою реализацию ViewModelProvider.Factory. По умолчанию используется DefaultFactory, которая является вложенным классом в ViewModelProviders.

Класс **ViewModelProvider,** класс, возвращающий инстанс ViewModel. Класс выполняет роль посредника с ViewModelStore, который, хранит и поднимает интанс ViewModel и возвращает его с помощью метода get, который имеет две сигнатуры get(Class) и get(String key, Class modelClass). Смысл заключается в том, что имеется возможность привязать несколько ViewModel к нашему Activity/Fragment даже одного типа. Метод get возвращает их по String key, который по умолчанию формируется как: «android.arch.lifecycle.ViewModelProvider.DefaultKey:» + canonicalName

Класс **ViewModelStores**, представляет собой фабричный метод. На данный момент, в пакете android.arch присутствует как один интерфейс, так и один подкласс ViewModelStore.

Класс **ViewModelStore**, класс хранит в себе HashMap<String, ViewModel>, методы get и put, соответственно, возвращают по ключу (по тому самому, который мы формируем во ViewModelProvider) или добавляют ViewModel. Метод clear(), вызовет метод onCleared() у всех созданных ViewModel которые добавил разработчик.

Поведение ViewModel из задания 4.8:

- activity вызывает метод модели getProgressState();
- модель создает MutableLiveData и стартует асинхронный процесс;
- activity подписывается на получение данных от модели LiveData;
- происходит поворот экрана. Модель никак не реагирует на данное событие и ждет завершение потока;
- activity пересоздается, получает ту же самую модель от провайдера, получает тот же самый LiveData от модели и подписывается на него и ждет данные;
- поток завершается и устанавливает значение ProgressBar, модель передает его в MutableLiveData;
- activity получает данные от LiveData.

Context - не стоит передавать Activity в модель в качестве Context. Это может привести к утечкам памяти. Если в модели требуется объект Context, то имеется возможность наследовать не ViewModel, а AndroidViewModel.

```
public class MyViewModel extends AndroidViewModel {

    public MyViewModel(@NonNull Application application) {
        super(application);
    }

    public void doSomething() {
        Context context = getApplication();
        // ....
    }
}
```

1.8.2 Передача данных между фрагментами

ViewModel может быть использована для передачи данных между фрагментами, которые находятся в одном Activity.

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.

```
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;
    }
}
```

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. A DetailFragment - подписывается и получает данные.

2 КОНТРОЛЬНОЕ ЗАДАНИЕ

Открыть контрольное задание, построенное на базе NavigationDrawer. Добавить фрагмент по работе с данными.

Придумать функцию по работе с базой данных Room и реализовать.