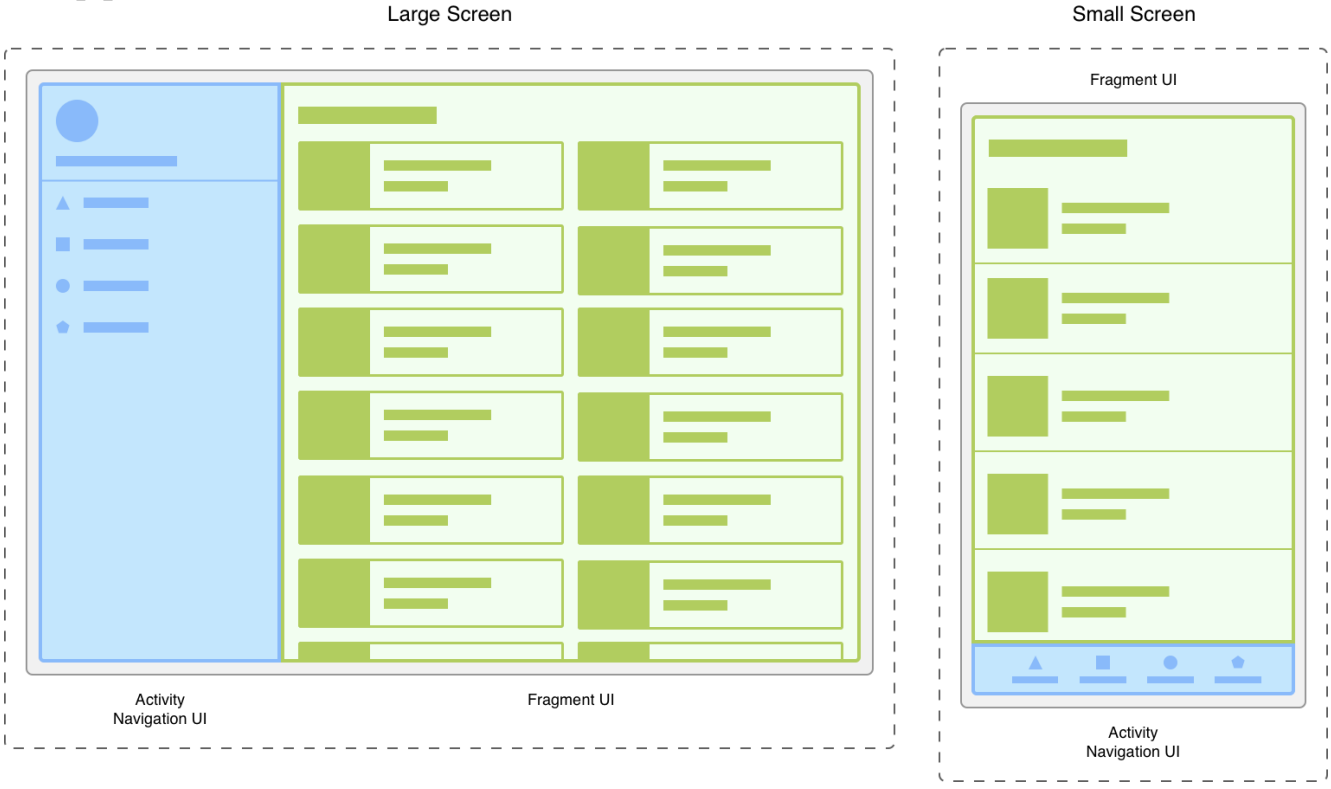
№9 Фрагменты (fragments)

Что такое фрагменты

https://developer.android.com/guide/components/fragments

Фрагмент представляет собой многократно используемую часть пользовательского интерфейса приложения. Фрагмент определяет и управляет своим собственным макетом, имеет собственный жизненный цикл и может обрабатывать свои собственные входные события. Фрагменты не могут существовать сами по себе - они должны размещаться в активности или другом фрагменте. Иерархия представления фрагмента становится частью иерархии представления хоста или присоединяется к ней.

Фрагменты впервые появились в Android версии 3.0 (API уровня 11), главным образом, для обеспечения большей динамичности и гибкости пользовательских интерфейсов на больших экранах, например, у планшетов. Поскольку экраны планшетов гораздо больше, чем у смартфонов, они предоставляют больше возможностей для объединения и перестановки компонентов пользовательского интерфейса.



Например, новостное приложение может использовать один фрагмент для показа списка статей слева, а другой—для отображения статьи справа. Оба фрагмента отображаются в одной activity рядом друг с другом, и каждый имеет собственный набор методов обратного вызова жизненного цикла и управляет собственными событиями пользовательского ввода. Таким образом, вместо применения одной activity для выбора статьи, а другой — для чтения статей, пользователь может выбрать статью и читать ее в рамках одной activity, как на планшете, изображенном на рисунке.

Фрагмент (класс Fragment) представляет поведение или часть пользовательского интерфейса в активности (класс Activity).

Фрагмент всегда должен быть встроен в activity, и на его жизненный цикл напрямую влияет жизненный цикл activity. Например, когда activity приостановлена, в том же состоянии находятся и все фрагменты внутри нее, а когда activity уничтожается, уничтожаются и все фрагменты. Однако пока activity выполняется можно манипулировать каждым фрагментом независимо, например добавлять или удалять их.

9.1 Жизненный цикл фрагмента

Для создания фрагмента необходимо создать подкласс класса Fragment (или его существующего подкласса). Класс **Fragment** (androidx.fragment.app.Fragment;) имеет код, во многом схожий с кодом Activity. Он содержит методы обратного вызова, аналогичные методам активности, такие как **onCreate(), onStart(), onPause() и onStop()** и реализует интерфейс *LifecycleOwner*. На практике, если требуется преобразовать существующее приложение Android так, чтобы в нем использовались фрагменты, достаточно просто переместить код из методов обратного вызова операции в соответствующие методы обратного вызова фрагмента.

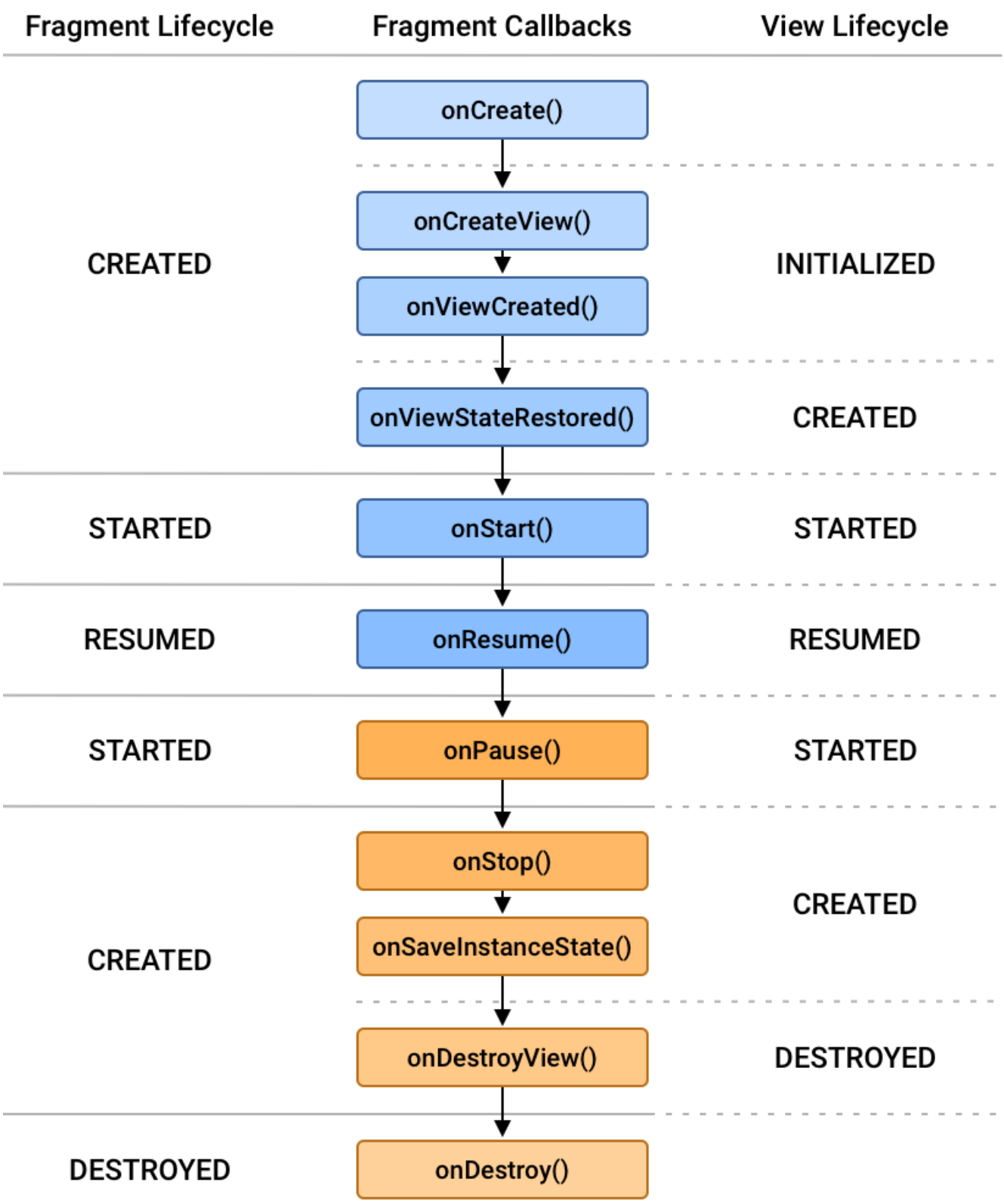
Фрагменты содержатся в активностях и находятся под их управлением.

Жизненный цикл фрагмента связан с жизненным циклом активности

Состояния фрагмента (определены в Lifecycle.State enum.):

INITIALIZED
CREATED
STARTED
RESUMED
DESTROYED

Фрагмент и его визуальный интерфейс или View имеет отдельный жизненный цикл.



Как правило, необходимо реализовать следующие методы жизненного цикла:

onCreate()

Система вызывает этот метод, когда создает фрагмент. В своей реализации разработчик должен инициализировать ключевые компоненты фрагмента, которые требуется сохранить, когда фрагмент находится в состоянии паузы или возобновлен после остановки.

onCreateView()

Система вызывает этот метод при первом отображении пользовательского интерфейса фрагмента на дисплее. Для прорисовки пользовательского интерфейса фрагмента следует возвратить из этого метода объект View, который является корневым в макете фрагмента. Если фрагмент не имеет пользовательского интерфейса, можно возвратить n

Вызывается для создания иерархии представлений, связанной с фрагментом

onPause()

Система вызывает этот метод как первое указание того, что пользователь покидает фрагмент (это не всегда означает уничтожение фрагмента). Обычно именно в этот момент необходимо фиксировать все изменения, которые должны быть сохранены за рамками текущего сеанса работы пользователя (поскольку пользователь может не вернуться назад).

В большинстве приложений для каждого фрагмента должны быть реализованы, как минимум, эти три метода. Однако существуют и другие методы обратного вызова, которые следует использовать для управления различными этапами жизненного цикла фрагмента:

onAttach()

Вызывается, когда фрагмент связывается с активностью (ему передается объект Activity).

onViewCreated()

Вызывается после создания представления фрагмента.

onActivityCreated()

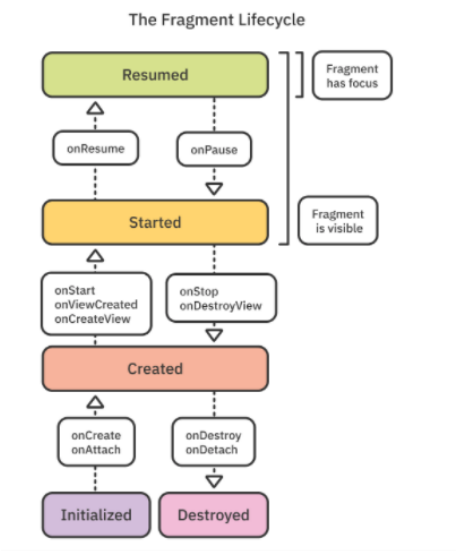
Вызывается, когда метод onCreate(), принадлежащий актиности, возвращает управление.

onDestroyView()

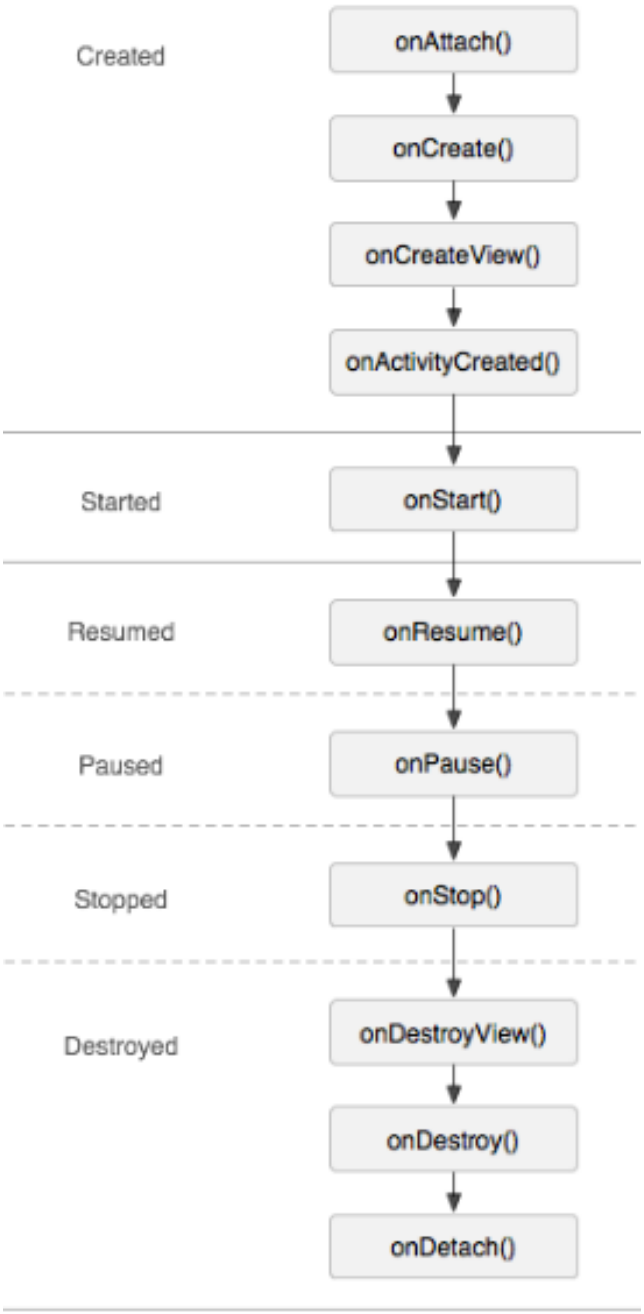
Вызывается при удалении иерархии представлений, связанной с фрагментом.

onDetach()

Вызывается при разрыве связи фрагмента с активностью.



Зависимость жизненого цикла фрагмента от содержащей его актисности иллюстрируется рисунком. Можно видеть, что очередное состояние активности определяет, какие методы обратного вызова может принимать фрагмент. Например, когда активность принимает метод обратного вызова **onCreate()**, фрагмент внутри этой активности принимает метод обратного вызова **onActivityCreated()**.



Когда активность переходит в состояние «возобновлена», можно свободно добавлять в нее фрагменты и удалять их. Таким образом, жизненный цикл фрагмента может быть независимо изменен, только пока операция остается в состоянии «возобновлена».

Однако, когда активность выходит из этого состояния, продвижение фрагмента по его жизненному циклу снова осуществляется актисностью. *Таким образом, Activity являеся хост и управляет фрагментом*.

Разработчик может сохранить состояние фрагмента с помощью **Bundle** на случай, если процесс активности будет уничтожен, а разработчику понадобится восстановить состояние фрагмента при повторном создании активности. Состояние можно сохранить во время выполнения метода обратного вызова **onSaveInstanceState()** во фрагменте и восстановить его во время выполнения onCreate(), onCreateView() или onActivityCreated().

По умолчанию активность помещается в управляемый системой *стек переходов назад*, когда она останавливается (чтобы пользователь мог вернуться к ней с помощью кнопки *Назад)*. В то же время, фрагмент помещается в стек переходов назад, управляемый активностью, только когда разработчик явно запросит сохранение конкретного экземпляра, вызвав метод **addToBackStack()** во время транзакции, удаляющей фрагментом.

В остальном управление жизненным циклом фрагмента очень похоже на управление жизненным циклом активности. Поэтому практические рекомендации по управлению жизненным циклом активностей применимы и к фрагментам.

Метод жизненного цикла	Активность?	Фрагмент?
onAttach()		✓
onCreate()	✓	✓
onCreateView()		✓
onActivityCreated()		✓
onStart()	✓	✓
onPause()	✓	✓
onResume()	✓	✓
onStop()	✓	✓
onDestroyView()		✓
onRestart()	✓	
onDestroy()	✓	✓
onDetach()		✓

9.2 Принципы работы с фрагментами

9.2.1. Разновидности и классы фрагментов

DialogFragment – представляет собой перемещаемое диалоговое окно. Дает возможность вставить диалоговое окно фрагмента в управляемый стек переходов назад, что позволяет пользователю вернуться к закрытому фрагменту.

ListFragment – предоставляет отображение списка элементов, управляемых адаптером *SimpleCursorAdapter*. Предоставляет несколько методов для управления списком.

PreferenceFragmentCompat– позволяет выполнять отображение иерархии объектов *Preference* в виде списка, аналогично классу *PreferenceActivity*.

Для транзакций (добавление, удаление, замена) используется класс-помощник *android.app.FragmentTransaction*. В 2018 году Google объявила фрагменты из пакета *androd.app* устаревшими. Поэтому их необходимо заменять классами из библиотеки совместимости:

- `androidx.fragment.app.FragmentActivity`
- `androidx.fragment.app.Fragment;`
- `androidx.fragment.app.FragmentManager;`
- `androidx.fragment.app.FragmentTransaction`

9.2.2. Добавление фрагментов в Activity

У каждого фрагмента должен быть свой класс. Класс наследуется от класса *Fragment* или схожих классов.

```
public class ExampleFragment  extends Fragment {
    public ExampleFragment() {
        super(R.layout.example_fragment);
    }
}
```

Разметку для фрагмента можно создать программно или декларативно через XML.Создание разметки для фрагмента ничем не отличается от создания разметки для активности.

XML

Для добавления фрагмента в layout используется *FragmentContainerView* , который определяет место размещения фрагмента. *FragmentContainerView* включает исправления и предпочтительней чем FrameLayout.

Чтобы декларативно добавить фрагмент в XML макет:

```
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/fragment_container_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:name="by.bstu.patsei.fragmentapp.ui.main.MainFragment" />
```

Атрибут *android:name* указывает имя класса фрагмента для создания экземпляра. Когда создается макет activity, создается экземпляр указанного фрагмента, вызывается onInflate ()и создается FragmentTransaction для добавления фрагмента в FragmentManager.

Программно

При программном создании name не указывается. FragmentTransaction используется для создания экземпляра фрагмента и добавления его в макет активности.

```
public class ExampleActivity extends AppCompatActivity {
    public ExampleActivity() {
        super(R.layout.example_activity);
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (savedInstanceState == null) {
            getSupportFragmentManager().beginTransaction()
                    .setReorderingAllowed(true)
                    .add(R.id.fragment_container_view, ExampleFragment.class,
null)
                    .commit();
        }
    }
}
```

Транзакция фрагмента создается только в том случае, если *saveInstanceState* имеет значение *null*. Это необходимо для того, чтобы фрагмент был добавлен только один раз при первом создании activity. Когда происходит изменение конфигурации и

активность создается повторно, значение savedInstanceState больше не равно нулю, и фрагмент не нужно добавлять во второй раз, он должен автоматически восстанавливается из *saveInstanceState*.

9.2.2 Установа начальных данных в фрагмент

Если фрагменту требуются некоторые начальные данные, аргументы могут быть переданы через *Bundle* при вызове *FragmentTransaction*.add:

```
public class ExampleActivity extends AppCompatActivity {
    public ExampleActivity() {
        super(R.layout.example_activity);
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (savedInstanceState == null) {
            Bundle bundle = new Bundle();
            bundle.putInt("Counter", 0);

            getSupportFragmentManager().beginTransaction()
                    .setReorderingAllowed(true)
                    .add(R.id.fragment_container_view, ExampleFragment.class,
bundle)
                    .commit();
        }
    }
}
```

Затем *Bundle* можно получить из фрагмента, вызвав *requireArguments*(), и соответствующие методы для получения каждого аргумента.

```
public class ExampleFragment  extends Fragment {
    public ExampleFragment() {
        super(R.layout.example_fragment);

    }

    @Override
    public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
        int someInt = requireArguments().getInt("Counter");

    }
}
```

9.2.3 Fragment Manager

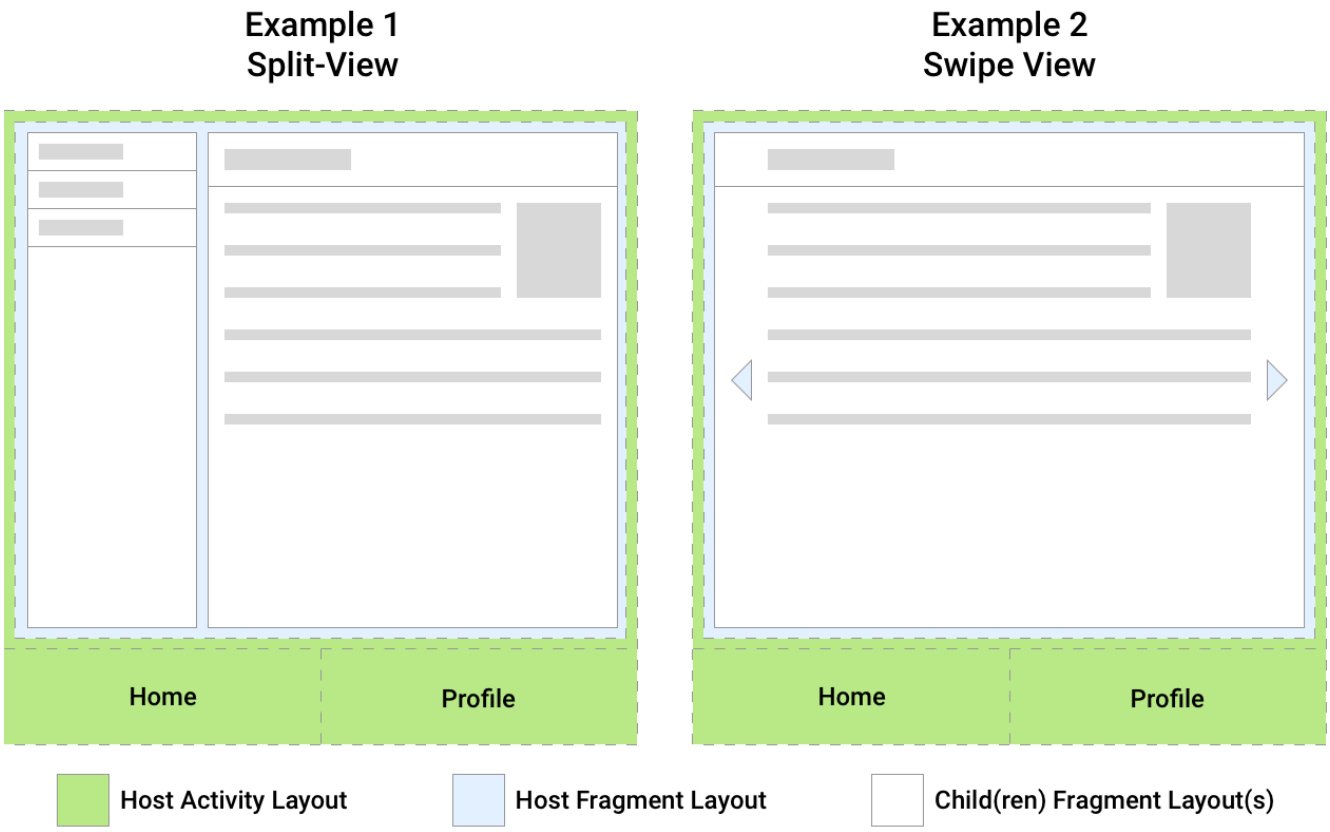
FragmentManager - класс, отвечающий действия с фрагментами добавление, удаление или замена , добавление с стек).

Получение доступа в activity

Каждая *FragmentActivity* (это суперкласс активности) и ее подклассы, такие как *AppCompatActivity*, имеют доступ к *FragmentManager* через метод ***getSupportFragmentManager ()***.

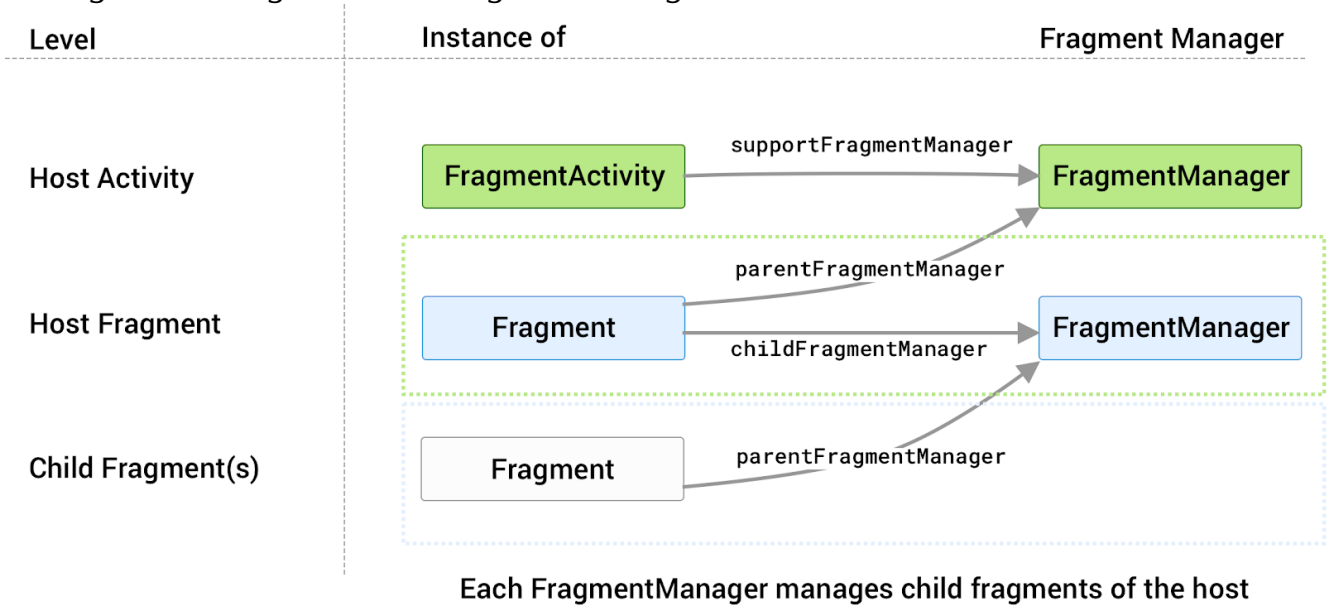
Получение доступа во фрагменте

Фрагменты также могут содержать один или несколько дочерних фрагментов.



Внутри фрагмента вы можете получить ссылку на *FragmentManager*, который управляет дочерними элементами фрагмента через ***getChildFragmentManager*** (). Если вам нужен доступ к его хосту *FragmentManager*, вы можете использовать ***getParentFragmentManager*** ().

Можно думать о каждом хосте как о связанном с ним *FragmentManager*, который управляет его дочерними фрагментами. Это показано на рисунке ниже вместе с сопоставлениями свойств между *supportFragmentManager*, *parentFragmentManager* и *childFragmentManager*.



Использование FragmentManager

FragmentManager управляет backstack фрагментов. Каждый набор изменений

фиксируется вместе как единое целое, называемое *FragmentTransaction*.

Когда пользователь нажимает кнопку «Back» или когда вызываете FragmentManager.popBackStack(), транзакция самого верхнего фрагмента извлекается из стека и транзакция отменяется.

```
FragmentManager fragmentManager = getSupportFragmentManager();
fragmentManager.beginTransaction()
        .replace(R.id.fragment_container_view, ExampleFragment.class, null)
        .setReorderingAllowed(true)
        .addToBackStack("name") // name can be null
        .commit();
```

ExampleFragment заменяет фрагмент, который в настоящее время находится в контейнере макета, идентифицированном идентификатором *R.id.fragment_container_view*.

setReorderingAllowed(true) оптимизирует изменения состояния фрагментов, участвующих в транзакции, чтобы анимация и переходы работали правильно.

addToBackStack() фиксирует транзакцию в стеке.. Если добавили или удалили несколько фрагментов в рамках одной транзакции, все они будут отменены при извлечении из backstack. Необязательное имя, указанное в вызове *addToBackStack*(), дает возможность вернуться к этой конкретной транзакции с помощью *popBackStack* ().

Поиск Fragment

Класс *FragmentManager* имеет два метода, позволяющих найти фрагмент, который связан с активностью:

- *findFragmentById*() (с UI): находит фрагмент по идентификатору;
- *findFragmentByTag*() (без UI): находит фрагмент по заданному тегу.

Вы можете получить ссылку на текущий фрагмент в контейнере макета, используя *findFragmentById*() либо по заданному идентификатору XML, либо по идентификатору контейнера при добавлении в *FragmentTransaction*:

```
 ExampleFragment fragment =
            (ExampleFragment)
fragmentManager.findFragmentById(R.id.fragment_container_view);
}
```

В качестве альтернативы можно присвоить фрагменту уникальный тег и получить ссылку с помощью *findFragmentByTag* () (тег задается с помощью XML-атрибута android: tag для фрагментов или во время операции add () или replace () в FragmentTransaction).

```
    FragmentManager fragmentManager = getSupportFragmentManager();
  fragmentManager.beginTransaction()
             .replace(R.id.fragment_container, ExampleFragment.class, null, "tag")
             .setReorderingAllowed(true)
             .addToBackStack(null)
             .commit();

...
```

```
            ExampleFragment fragment = (ExampleFragment)
fragmentManager.findFragmentByTag("tag");
```

Множественный Backstack

В некоторых может потребоваться поддержка нескольких backstack. Например, приложение использует нижнюю панель навигации. *FragmentManager* позволяет поддерживать несколько backstack с помощью методов ***saveBackStack*** () и ***restoreBackStack*** (). Эти методы позволяют переключаться между теками, сохраняя один и восстанавливая другой.

Например, предположим, что добавили фрагмент в заднийв стек, зафиксировав *FragmentTransaction* с помощью *addToBackStack* ():

```
getSupportFragmentManager().beginTransaction()
        .replace(R.id.fragment_container, ExampleFragment.class, null)
        .setReorderingAllowed(true)
        .addToBackStack("replacement")
        .commit();
```

По умолчанию FragmentManager использует FragmentFactory, предоставляемый платформой, для создания нового экземпляра фрагмента. Эта фабрика по умолчанию. Но нельзя использовать эту фабрику для предоставления зависимостей фрагменту. Это также означает, что любой пользовательский конструктор, который вы использовали для создания фрагмента не используется во время его воссоздания.

Чтобы обеспечить зависимости для о фрагмента или использовать любой настраиваемый конструктор, нужно создать пользовательский подкласс *FragmentFactory*, а затем переопределить *FragmentFactory*.*instantiate*. Затем можно заменить фабрику по умолчанию *FragmentManager* своей фабрикой.

https://developer.android.com/guide/fragments/fragmentmanager

9.2.4 Fragment Transactions

FragmentManager может добавлять, удалять, заменять и выполнять другие действия с фрагментами. Каждый набор изменений фрагментов называется транзакцией. Можно указать, что делать внутри транзакции, используя API, предоставляемые классом *FragmentTransaction*. Можно сгруппировать несколько действий в одну транзакцию, что полезно когда у вас есть несколько одноуровневых фрагментов, отображаемых на одном экране, например, с разделенными представлениями.

Каждую транзакцию можно сохранить в стеке, что позволит пользователю перемещаться по изменениям фрагмента - аналогично перемещению по активностям.

Получить экземпляр *FragmentTransaction* из *FragmentManager* можно вызвав *beginTransaction* (), как показано в следующем примере:

```
FragmentManager fragmentManager = getSupportFragmentManager();
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
```

Последний вызов каждой *FragmentTransaction* должен зафиксировать транзакцию. Вызов *commit* () сообщает *FragmentManager*, что все операции были добавлены в транзакцию.

```
FragmentManager fragmentManager = getSupportFragmentManager();
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
fragmentTransaction.commit();
```

Для совместимости поведения флаг переупорядочения по умолчанию не включен. Однако требуется, чтобы *FragmentManager* правильно выполнял *FragmentTransaction*, особенно когда он работает с backstack и запускает анимацию и переходы. Включение флага гарантирует, что, если несколько транзакций выполняются вместе, любые промежуточные фрагменты не претерпевают изменений жизненного цикла или не выполняют их анимацию или переходы.

```
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction()
    //...
        .setReorderingAllowed(true)
        .commit();
```

Для создания динамического UI используют разные методы управления транзакций:

— *add*(): добавляет фрагмент к активности;
— *remove*(): удаляет фрагмент из активности;
— *replace*(): заменяет один фрагмент на другой;
— *hide*(): прячет фрагмент (делает невидимым на экране);
— *show*(): выводит скрытый фрагмент на экран;
— *detach*() (API 13): отсоединяет фрагмент от графического интерфейса, но экземпляр класса сохраняется;
— *attach*() (API 13): присоединяет фрагмент, который был отсоединён методом *detach*().

Методы *remove*(), *replace*(), *detach*(), *attach*() не применимы к статичным фрагментам.

Например:

```
FragmentManager fragmentManager = getFragmentManager();

fragmentManager.beginTransaction()
    .remove(fragment1)
    .add(R.id.fragment_container, fragment2)
    .show(fragment3)
    .hide(fragment4)
    .commit();
```

По умолчанию изменения, внесенные в *FragmentTransaction*, не добавляются в задний стек. Чтобы сохранить эти изменения, вы можете вызвать *addToBackStack*() для *FragmentTransaction*.

Вызов *commit*() не выполняет транзакцию немедленно. Транзакция планируется для запуска в основном потоке пользовательского интерфейса, как

только это будет возможно. Однако при необходимости вы можете вызвать *commitNow*(), чтобы немедленно запустить транзакцию фрагмента в потоке пользовательского интерфейса.

Обратите внимание, что *commitNow* несовместим с *addToBackStack*. В качестве альтернативы можете выполнить все ожидающие транзакции *FragmentTransactions*, отправленные вызовами *commit*(), которые еще не были выполнены, путем вызова *executePendingTransactions*(). Этот подход совместим с *addToBackStack*.

Для подавляющего большинства случаев лучше подходит *commit*().

9.2.5 Коммуникации фрагментов

Существует 4 способа коммуникаций между фрагментами

- **Interface**
- **ViewModel**
- **RxJava**
- **Event Bus**

9.3 Пример работы с фрагментами

9.3.1Создание фрагмента

Рассмтрим создание фрагментов на примере. При запуске приложение открывает активность *MainActivity*. Активность использует два фрагмента *StudentListFragment* и *StudentDetailFragment***.** Фрагмент *StudentListFragment* отображает список студентов. Фрагмент *StudentDetailFragment* отображает подробное описание студента. Оба фрагмента получают свои данные из *Student.java.*

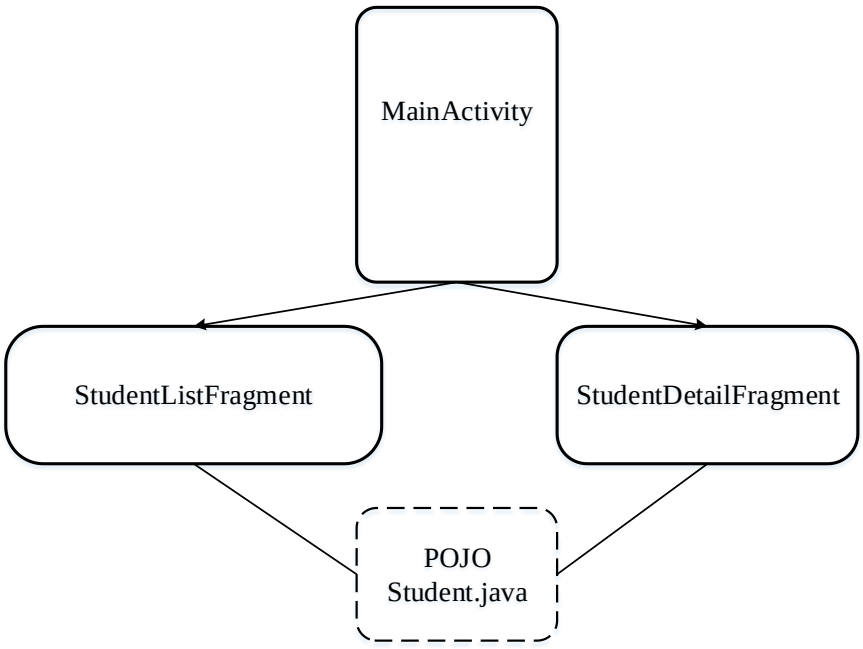


Рис. Схема приложения с фрагментами

В приложении это это может выглядеть так :

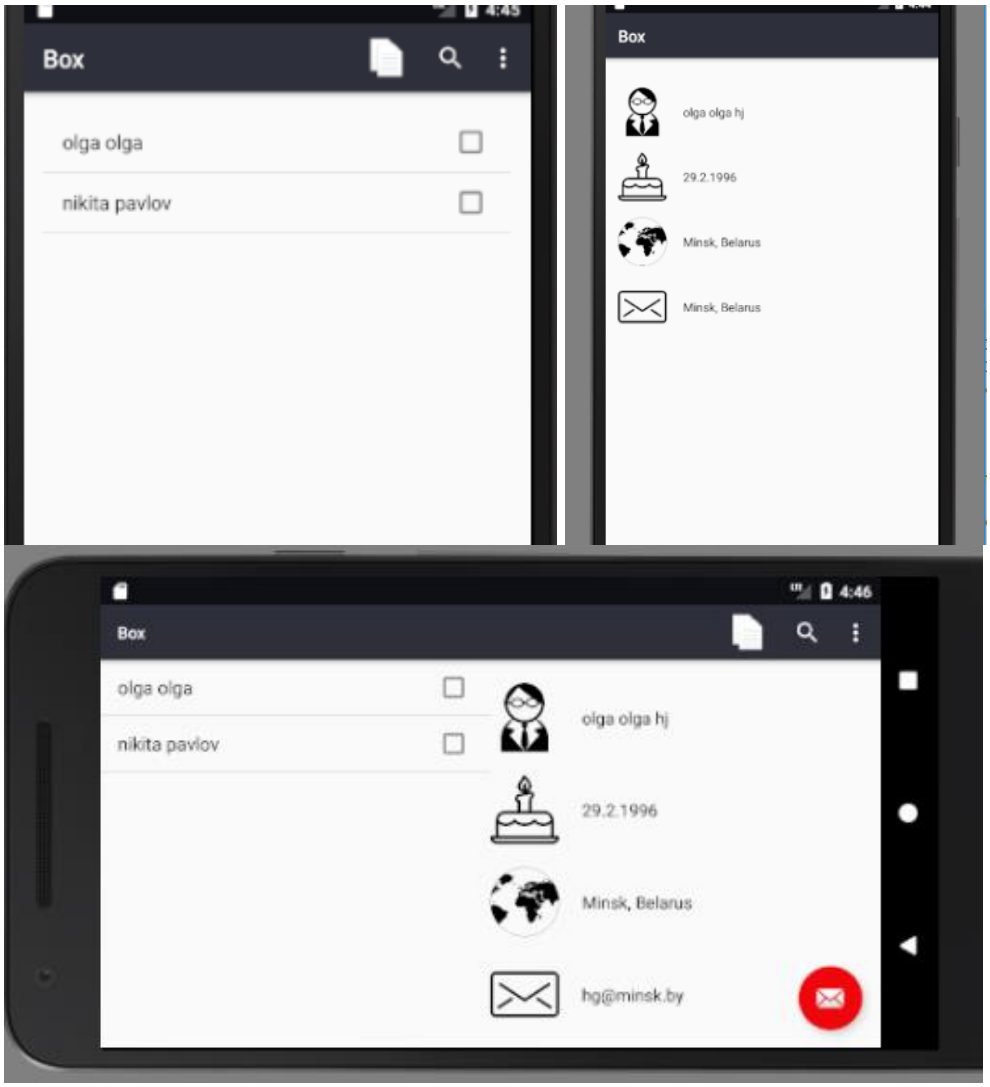


Рис. Отображение приложения с фрагментами для разной ориентации

Изменим приложение так, чтобы оно по-разному выглядело и работало в зависимости от типа устройства, на котором оно выполняется. Если приложение запущено на устройстве с большим экраном, то фрагменты будут размещаться рядом друг с другом. На устройствах с малыми экранами фрагменты будут находиться в разных активностях.

Начнем с добавления класса *Student* в приложение. Это обычный файл модели, из которого приложение получает информацию о студентах. Класс определяет статический массив из трех элементов, как источник информации:

```
public class Student {
    private String name;
    private String info;
    private int rate;

    public static final Student[] studList = {
    new Student("Navichik", "Address tel course group university", 9),
    new Student("Mihalkov", "Address tel course group university", 4),
    new Student("Malishev", "Address tel course group university", 7)
    };
```

```
    private Student(String name, String info, int rate) {
        this.name = name;
        this.info = info;
        this.rate = rate;
    }
    // ..
```

Теперь добавьте в проект новый фрагмент с именем *StudentDetailFragment*, предназначенный для вывода подробной информации о студенте. Новые фрагменты добавляются примерно так же, как и новые активности. В Android Studio выберите команду *File→New...→Fragment→Fragment (Blank)*. Вам будет предложено задать параметры нового фрагмента. Присвойте фрагменту имя и присвойте макету фрагмента имя *fragment_student_detail*

9.3.2 Макет фрагмента

Когда фрагмент добавлен как часть макета активноси, он находится в объекте *ViewGroup* внутри иерархии представлений активности и определяет собственный макет. Разработчик может вставить фрагмент в макет активности двумя способами. Для этого следует раньше объявлялся фрагмент в файле макета активности как элемент *<fragment>* или использовать androidx.fragment.app.FragmentContainerView. Можно также использовать фрагмент без интерфейса в качестве невидимого рабочего потока активности.

Начнем с обновления разметки макета фрагмента. Откройте и определите файл *fragment_student_detail.xml:*

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="by.bstu.pnv.studentfragments.StudentDetailFragment"
    android:orientation="vertical">

    <!-- TODO: Update blank fragment layout -->

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_blank_fragment" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="30dp"
        android:id="@+id/name"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="25dp"
        android:id="@+id/info"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="30dp"
        android:id="@+id/rate"/>
```

```
</LinearLayout>
```

Разметка макета фрагмента почти не отличается от разметки макета активности. Макет прост и состоит из четырех надписей.

Создавая макеты фрагментов для своих приложений, вы можете использовать в них все представления и макеты, которые уже использовались для построения активностей. Перейдем к коду самого фрагмента.

9.3.3 Код фрагмента

Замените код, сгенерированный Android Studio, следующим кодом.

```
public class StudentDetailFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState) {

        return inflater.inflate(R.layout.fragment_student_detail,
                                                container,
                                                false);
    }
}
```

Метод *onCreateView*() вызывается тогда, когда потребуется макет фрагмента. В качестве параметра ему передается объект *ViewGroup* активности, в который должен быть вставлен макет фрагмента. Метод *inflater.inflate*() является аналогом метода *setContentView*() только для фрагментов. Аргумент *R.layout.fragment_student_detail* определяет какой макет используется фрагментом.

У каждого фрагмента должен быть определен открытый конструктор без аргументов. Android использует его для повторного создания экземпляра в случае необходимости, и при отсутствии такого конструктора выдается исключение времени выполнения. На практике добавлять такой конструктор в код фрагмента нужно лишь в том случае, если вы включаете другой конструктор с одним или несколькими аргументами.

9.3.4 Добавление фрагмента в макет активности

Когда создавали проект, среда создала активность с именем *MainActivity.java* и макет с именем *activity_main.xml*. Изменим макет так, чтобы он содержал только что созданный фрагмент.

```
<fragment
        class="by.bstu.pnv.studentfragments.StudentDetailFragment"
        android:id="@+id/detail_frag"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</RelativeLayout>
```

Макет состоит из единственного элемента <*fragment*>. Элемент <*fragment*> используется для добавления фрагмента в макет активности. Чтобы указать, какой именно фрагмент должен использоваться, вы присваиваете атрибуту *class* полное имя фрагмента. В нашем примере создается фрагмент *StudentDetailFragment*.

9.3.5 Обновление View фрагмента

Активность должна взаимодействовать с фрагментом. Так активность сообщает фрагменту, данные какой записи в нем должны выводиться.

Для этого мы добавим в фрагмент простой метод, который будет задавать значение идентификатора:

```
public class StudentDetailFragment extends Fragment {
    private long studentId;

//...

public void setStudent(long id) {
    this.studentId = id;
}
```

Активность будет использовать этот метод для передачи идентификатора фрагменту. Позднее в зависимости от идентификатора будут заполняться представления.

9.3.6 Заполнение представлений в методе onStart() фрагмента

Класс *StudentDetailFragment* должен обновить свои представления подробной информацией о студенте. Это необходимо сделать при запуске активности, поэтому мы воспользуемся методом *onStart*() фрагмента:

```
@Override
public void onStart() {

    super.onStart();

    View view = getView();

    if (view != null) {
       TextView title = (TextView) view.findViewById(R.id.name);
       Student stud = Student.studList[(int) studentId];
       title.setText(stud.getName());
       TextView description = (TextView) view.findViewById(R.id.info);
       description.setText(stud.getInfo());
    }
}
```

Фрагменты отличаются от активностей и поэтому не поддерживают некоторые методы. Например, у фрагментов отсутствует метод *findViewById*(). Чтобы получить ссылку на представления фрагмента, сначала необходимо получить ссылку на корневое представление фрагмента методом *getView*() и по этой ссылке найти дочерние представления.

9.3.7 Взаимодействие активности и фрагмента

Прежде чем активность сможет взаимодействовать с фрагментом, она должна сначала получить ссылку на него. Для получения ссылки на фрагмент следует получить ссылку на ***диспетчера фрагментов*** активности при помощи метода *getFragmentManager*(). Затем метод *findFragmentById*() используется для получения ссылки на фрагмент:

```
public class MainActivity extends AppCompatActivity {

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

            StudentDetailFragment fragm =
                        (StudentDetailFragment)getFragmentManager().
                              findFragmentById(R.id.detail_frag);

      fragm.setStudent(1);
    }
}
```

Диспетчер фрагментов управляет всеми фрагментами, используемыми активностью.

Как видите, получение ссылки на фрагмент происходит после вызова *setContentView*(). Это важно, потому что до этого момента фрагмент еще не был создан. Вызов *fragm.setStudent*(1) сообщает фрагменту, о каком фрагменте нужно вывести подробную информацию. Пока мы просто задаем конкретный идентификатор в методе *onCreate*() активности, чтобы увидеть на экране хоть какие-нибудь данные.

Можно запустить приложение и протестировать. При запуске приложения создается активность *MainActivity*. *MainActivity* передает идентификатор студента классу *StudentDetailFragment* в методе *onCreate*(), вызывая метод *setStudent*() фрагмента. Фрагмент использует полученное значение в методе *onStart*() для заполнения надписей.

В нашем приложении надо создать новую транзакцию фрагмента, включить в нее одну операцию *add*, а затем закрепить:

```
fm.beginTransaction()
        .add(R.id.fragment_container, details)
        .commit();
```

9.3.9 Создание фрагмента со списком

Теперь необходимо создать второй фрагмент со списком студентов. После этого фрагменты можно будет использовать для создания разных пользовательских интерфейсов. Мы создадим фрагмент, содержащий списковое представление, и заполним его.

ListFragment – специализированный фрагмент, разновидность *Fragment*, для работы со списковым представлением. В макете по умолчанию этого фрагмента содержится компонент *ListView***.**

Как и списковая активность, такой фрагмент автоматически связывается со списковым представлением, и вам не придется создавать его самостоятельно. Списковые фрагменты определяют свой макет на программном уровне. Для обращения к ним из кода активности используется метод *getListView*(). Такое обращение необходимо для того, чтобы вы могли задать данные, которые должны выводиться в представлении.

Второе, вам не нужно реализовать собственного слушателя событий. Класс *ListFragment* регистрируется как слушатель события для спискового предстaлвения и отслеживает щелчки. Чтобы фрагмент реагировал на щелчки, достаточно реализовать метод *onListItemClick*().

Списковые фрагменты добавляются в проект точно так же, как и обычные фрагменты. Выберите команду *File→New...→Fragment→Fragment (Blank).* Присвойте фрагменту имя *StudentListFragment*, снимите флажок создания *xml* макета, а также флажок включения фабричных методов и интерфейсных обратных вызовов:

```
public class StudentListFragment extends ListFragment {

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container,  Bundle
savedInstanceState) {
   return super.onCreateView(inflater, container, savedInstanceState);
    }
}
```

Приведенный выше код создает простейший списковый фрагмент. Он должен расширять класс *ListFragment* вместо *Fragment*. Метод *onCreateView*() не является обязательным. Этот метод вызывается при создании представления фрагмента.

9.3.10 Использование ArrayAdapter для заполнения ListView

Для связывания данных со списковым представлением можно воспользоваться адаптером. *ListView* расширяет *AdapterView*, и именно этот класс обеспечивает работу представления с адаптерами. Так как информация в *StudentListFragment* будет передаваться в массиве названий, мы воспользуемся адаптером массива для связывания данных со списковым представлением:

```
ArrayAdapter<DataType> listAdapter = new ArrayAdapter<DataType>(
      context,
      android.R.layout.simple_list_item_1,
      array);
```

здесь *DataType* – тип данных, *array* – массив, а *context* – текущий контекст.

Класс *Fragment* не является субклассом *Context*, так что *this* не подходит. Вместо этого текущий контекст придется получать другим способом. Если адаптер используется в методе *onCreateView*() фрагмента, как в нашем примере, для получения контекста используется метод *getContext*() объекта *LayoutInflator*:

Когда адаптер будет создан, его следует связать с *ListView* при помощи метода *setListAdapter*() фрагмента. Внесем изменения:

```
public class StudentListFragment extends ListFragment {

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {

      String[] names = new String[Student.studList.length];
      for (int i = 0; i < names.length; i++) {
          names[i] = Student.studList[i].getName();
     }
      ArrayAdapter<String> adapter = new ArrayAdapter<String>(
inflater.getContext(), android.R.layout.simple_list_item_1,  names);
      setListAdapter(adapter);

   return super.onCreateView(inflater, container, savedInstanceState);
  }
}
```

Результат работы представлен на рис.

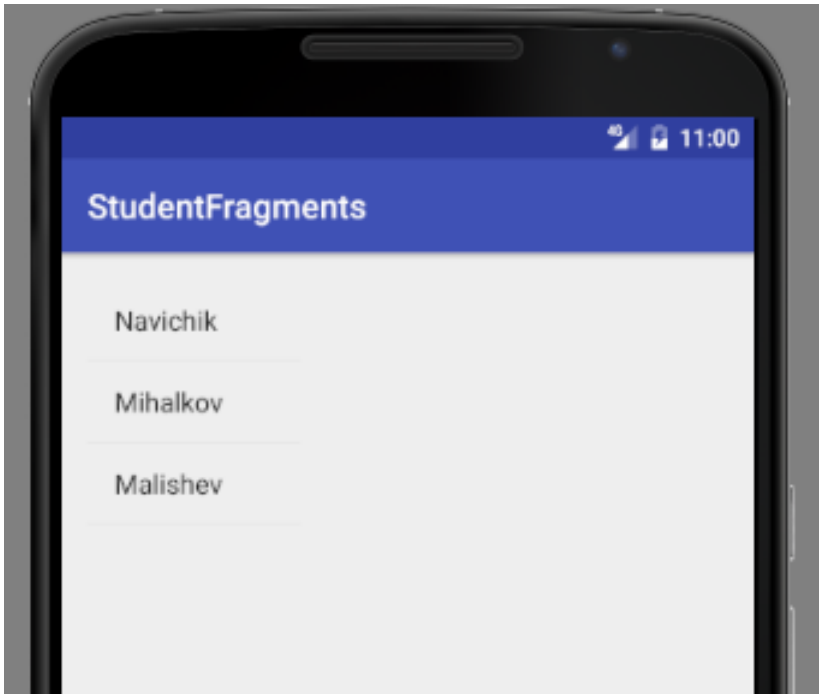


Рис. Фрагмент со списком

9.3.11 Включение StudentListFragment в макет

Добавим созданный фрагмент *StudentListFragment* в макет *MainActivity* так, чтобы он отображался слева от *StudentDetailFragment***.** Размещение фрагментов рядом друг с другом – типичный вариант дизайна приложений. Чтобы добиться нужного результата, мы воспользуемся линейным макетом с горизонтальной ориентацией. Для управления распределением горизонтального пространства между фрагментами будет использована система весов:

```
<LinearLayout>
    <fragment
         class="by.bstu.pnv.studentfragments.StudentListFragment"
         android:id="@+id/list_frag"
         android:layout_width="0dp"
         android:layout_weight="1"
         android:layout_height="match_parent"/>
    <fragment
         class="by.bstu.pnv.studentfragments.StudentDetailFragment"
         android:id="@+id/detail_frag"
         android:layout_width="0dp"
         android:layout_weight="2"
         android:layout_height="match_parent" />
</LinearLayout>
```

Визуально получим следующее

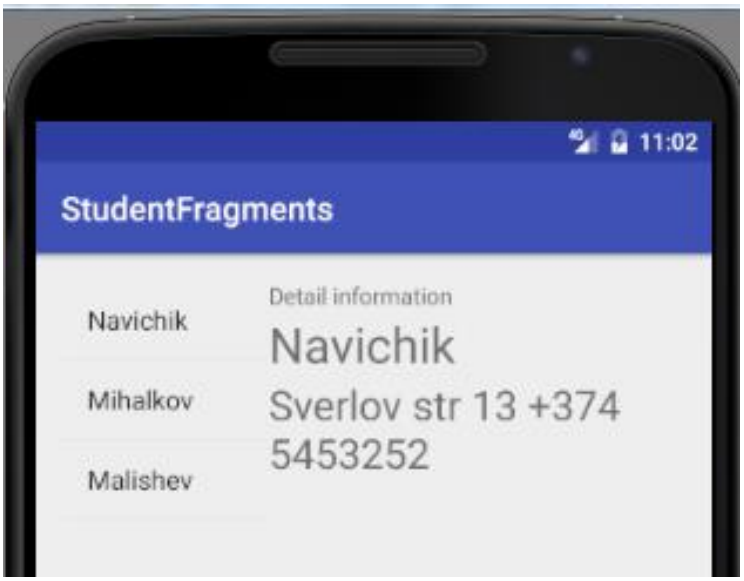


Рис. Отображение двух фрагментов

9.3.12 Связывание списка с детализацией

Для того, чтобы подробная информация изменялась при щелчке на варианте в списке есть несколько возможных решений.

Из-за возможности повторного использования наши фрагменты должны располагать минимумом информации о среде, содержащей их. Чем больше фрагмент знает об активности, использующей его, тем меньше он пригоден для повторного использования.

Мы хотим, чтобы фрагмент и активность взаимодействовали, располагая минимумом информации друг о друге. В Java для решения подобных задач используются *интерфейсы*. При определении интерфейса формулируются минимальные требования к объекту для его осмысленного взаимодействия с другим объектом. Это означает, что фрагмент сможет взаимодействовать практически с любой активностью — при условии, что эта *активность реализует необходимый интерфейс*.

Во время выполнения будет происходить следующая последовательность событий:

1) объект интерфейса сообщает фрагменту о своем желании прослушивать события щелчков;

2) пользователь делает выбор в списке;

3) вызывается метод *onListItemClicked*() в списковом фрагменте;

4) метод вызывает метод интерфейса с передачей идентификатора варианта.

Взглянув на схему жизненного цикла фрагмента, вы увидите, что при присоединении фрагмента к активности вызывается метод *onAttach*() фрагмента с передачей объекта активности. Этот метод можно использовать для регистрации активности во фрагменте. Рассмотрим код:

```
public class StudentListFragment extends ListFragment {

static interface StudentListListener {
        void itemClicked(long id);
};
    private StudentListListener listener;

@Override
public void onAttach(Activity activity) {
        super.onAttach(activity);
        this.listener = (StudentListListener)activity;
}
@Override
public void onListItemClick(ListView l, View v, int position, long id) {
        if (listener != null) {
            listener.itemClicked(id);
        }
}
```

Теперь активность *MainActivity.java* должна реализовать только что созданный интерфейс:

```
public class MainActivity extends AppCompatActivity
         implements StudentListFragment.StudentListListener  {

    @Override
    public void itemClicked(long id) {

    }

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

            StudentDetailFragment fragm =
(StudentDetailFragment)getFragmentManager().
findFragmentById(R.id.detail_frag);
fragm.setStudent(1);
    }
}
```

Зачеркнутые строки удаляются. Они не нужны. Фрагмент детализации будет заменяться новым фрагментом детализации каждый раз, когда потребуется изменить его содержимое.

9.3.13 Поддержка кнопки возврата. Back stack

Предположим, пользователь выбирает одного студента, а потом другого. При нажатии кнопки «*Назад*» он рассчитывает вернуться к первому из выбранных им студентов.

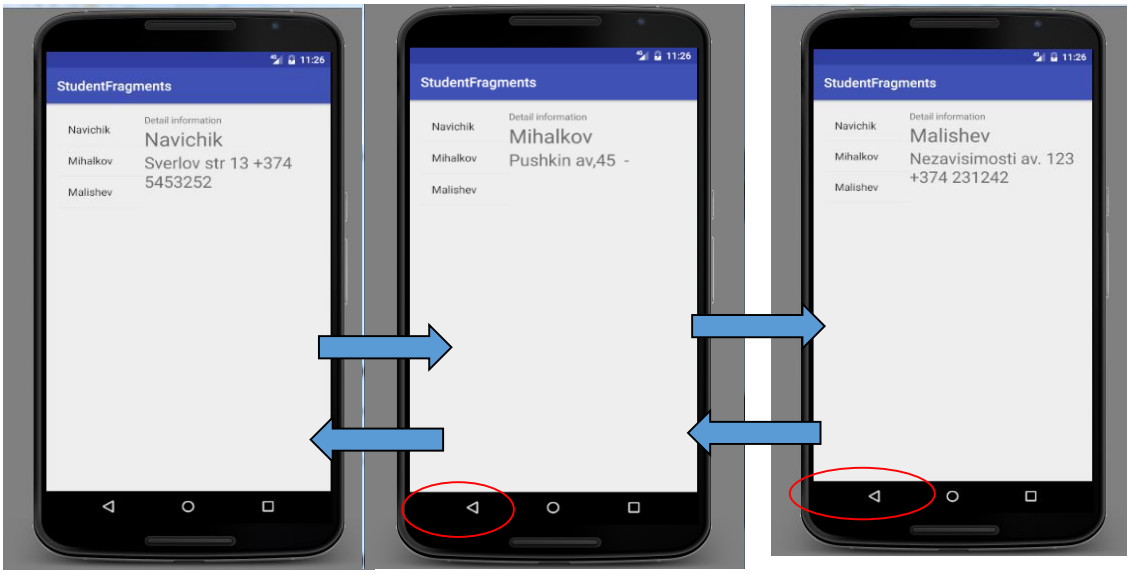


Рис. Использование кнопки *Назад* для фрагментов

Стек возврата (Back stack) представляет собой список «мест», посещенных на устройстве. Каждое «место» представлено транзакцией в стеке возврата. Многие транзакции осуществляют переход от одной активности к другой. Если нажать кнопку «*Назад*», транзакция будет отменена, и вы вернетесь к предыдущей активности. Однако транзакции в стеке возврата не обязаны быть переходами между активностями. Они могут быть простыми изменениями фрагментов на экране. Это означает, что смена фрагментов также может отменяться кнопкой «*Назад*», как и смена активностей.

Вместо того, чтобы обновлять представления в *StudentDetailFragment*, мы заменим весь фрагмент новым экземпляром *StudentDetailFragment*, настроенным для вывода информации о следующем выбранном студенте. При таком подходе замена фрагмента будет зарегистрирована в стеке возврата, а пользователь сможет отменить изменение нажатием кнопки «*Назад*».

Начать следует с внесения изменений в файл макета *activity_main.xml*. Вместо того, чтобы вставлять фрагмент напрямую, мы воспользуемся фреймом:

```
    …
<!--<fragment-->
<!--class="by.bstu.pnv.studentfragments.StudentDetailFragment"-->
<!--android:id="@+id/info_frag"-->
<!--android:layout_width="0dp"-->
<!--android:layout_weight="2"-->
<!--android:layout_height="match_parent" />-->
<fragment
        class="by.bstu.pnv.studentfragments.StudentListFragment"
        android:id="@+id/list_frag"
        android:layout_width="0dp"
        android:layout_weight="1"
```

```
        android:layout_height="match_parent"/>

    <FrameLayout
        android:id="@+id/fragment_container"
        android:layout_width="0dp"
        android:layout_weight="2"
        android:layout_height="match_parent" />

</LinearLayout>
```

Фрейм – разновидность группы представлений, используемая для резервирования области экрана. Он определяется элементом <*FrameLayout*> и используется для отображения одиночных объектов – в нашем примере это фрагмент. Мы помещаем фрагмент во фрейм, чтобы иметь возможность управлять его содержимым на программном уровне. Каждый раз, когда пользователь выбирает новый вариант в списковом представлении *StudentListFragment*, текущее содержимое фрейма заменяется новым экземпляром *StudentDetailFragment*

Замена фрагмента во время выполнения происходит в виде транзакции фрагмента – набора изменений, относящихся к фрагменту, которые должны применяться как единое целое:

```
StudentDetailFragment details = new  StudentDetailFragment();
FragmentTransaction ft = getFragmentManager().beginTransaction();
```

Затем указываются все действия, которые должны быть сгруппированы в транзакции. В нашем случае требуется заменить фрагмент во фрейме:

```
ft.replace(R.id.fragment_container, details);
```

Можно добавить фрагмент в контейнер или удалить:

```
ft.add(R.id.fragment_container, details);
ft.remove(details);
```

Метод *setTransition*() используется для определения анимации перехода, сопровождающей транзакцию. Допустимые значения – *TRANSIT_FRAGMENT_CLOSE* (фрагмент удаляется из стека), *TRANSIT_FRAGMENT_OPEN* (фрагмент добавляется), *TRANSIT_FRAGMENT_FADE* (фрагмент растворяется или проявляется) и *TRANSIT_NONE* (анимация отсутствует).

После определения всех действий, которые должны выполняться в составе транзакции, вызов метода *addToBackStack*() помещает транзакцию в стек возврата. Это делается для того, чтобы пользователь мог вернуться к предыдущему состоянию фрагмента нажатием кнопки «*Назад*». Метод *addToBackStack*() получает один параметр — строку с именем, используемую для идентификации транзакции. В большинстве случаев получать транзакцию вам не придется, поэтому при вызове передается *null*. Чтобы закрепить изменения в активности, вызовите метод *commit*():

```
ft.addToBackStack(null);
ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
```

```
ft.commit();
```

Полный код реализации выглядит так:

```
public class MainActivity extends AppCompatActivity
        implements StudentListFragment.StudentListListener  {

  @Override
  public void itemClicked(long id) {

     StudentDetailFragment details = new  StudentDetailFragment();

     FragmentTransaction ft = getFragmentManager().beginTransaction();
     details.setStudent(id);
     ft.replace(R.id.fragment_container, details);

     ft.addToBackStack(null);
     ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
     ft.commit();

    }
```

Запустите приложение и проверьте кнопку возврата ().

Рис. Проверка работы кнопки «*Назад*» для фрагментов

Если повернуть устройство, возникает проблема. Какого бы студента вы ни выбрали, при повороте приложение всегда выводит информацию о нулевом студенте (рис.).

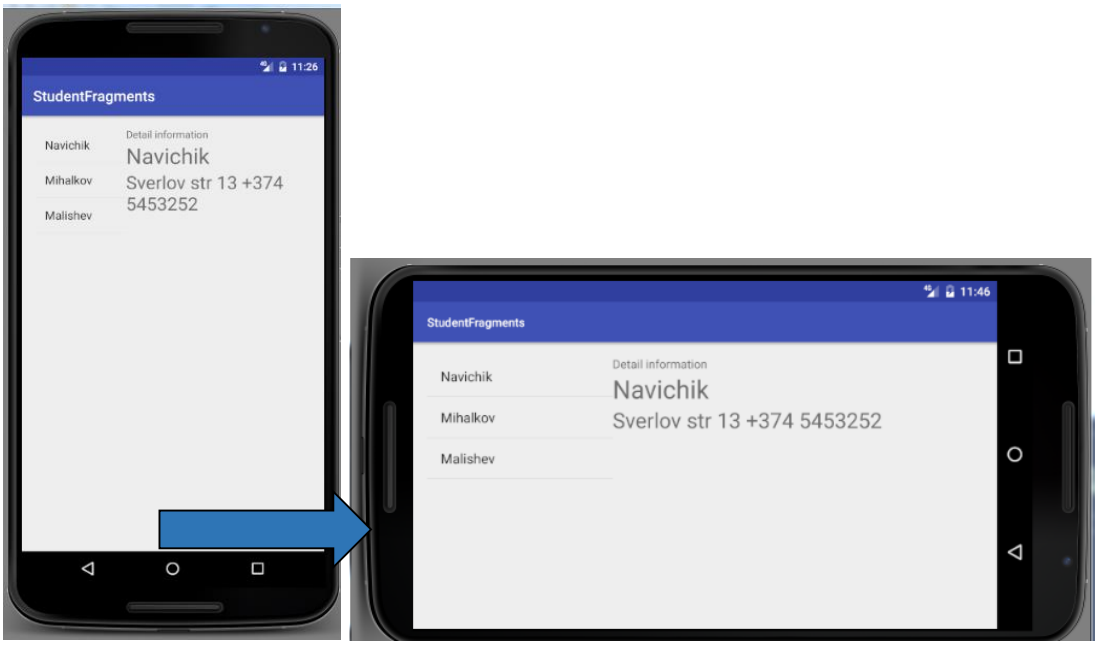


Рис. Работа фрагметов при повороте утройства

Когда рассматривали жизненный цикл активностей, вы узнали, что при повороте устройства Android уничтожает активность и создает ее заново. При этом значения локальных переменных, используемых активностью, могут быть потеряны. Если активность использует фрагмент, то фрагмент уничтожается и создается заново вместе с активностью. Это означает, что все локальные переменные, используемые фрагментом, тоже могут потерять свое состояние.

Для фрагментов эта проблема решается примерно так же, как и для активностей. Сначала переопределяете метод *onSaveInstanceState*() фрагмента и помещаете локальную переменную, значение которой требуется сохранить, в параметр *Bundle* метода.

После этого значение извлекается из *Bundle* в методе *onCreateView*() фрагмента:

```
public class StudentDetailFragment extends Fragment {
    private long studentId;

    @Override
    public void onSaveInstanceState(Bundle savedInstanceState) {
        savedInstanceState.putLong("stId", studentId);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {

    if (savedInstanceState != null) {
            studentId = savedInstanceState.getLong("stId");
   }
   return inflater.inflate(R.layout.fragment_student_detail, container, false);
    }
```

Итак, была реализована схема с фрагментами

Однако, если надо, чтобы фрагменты выводить в разных активностях, то схема взаимодействия должна другая

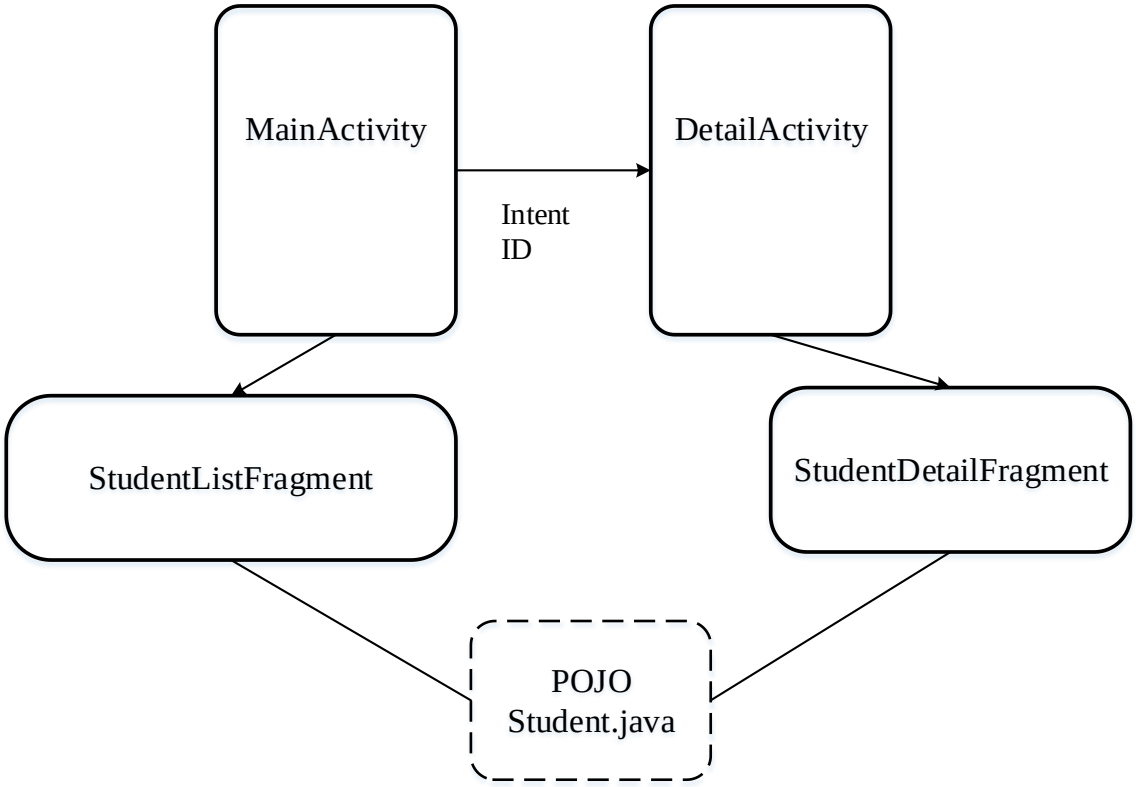


Рис. Схема взаимодействия фрагментов в активности

Приложение должно выглядеть и работать по-разному в зависимости от того, где оно выполняется — на телефоне или на планшете (разные размеры экранов).

Если вы хотите создать один макет для устройств с большим экраном и еще пару макетов для других устройств, макет для устройств с большим экраном помещается в папку *app/src/main/res/layout-large,* а макеты для других устройств — в папку *app/src/main/res/layout.*

Версия *activity_main.xml* из папки layout не содержит фрейм *fragment_container*, в отличие от версии *activity_main.xml* из папки *layout-large*. Фрагмент *StudentDetailFragment* должен отображаться только версией *activity_main.xml* из папки *layout-large*.

Обе ситуации можно обработать в *MainActivity*; для этого следует проверить, какой макет используется устройством. Такая проверка может быть выполнена поиском представления с идентификатором *fragment_container*. Если *fragment_container* существует, значит, устройство использует *activity_main.xml* из папки *layout-large*, следовательно, при выборе студента следует отобразить новый экземпляр *StudentDetailFragment*. Если же *fragment_container* не существует, то устройство использует версию *activity_main.xml* из папки *layout*, поэтому вместо этого запускается *DetailActivity*. Ниже приведен полный код *MainActivity.java*

```
public class MainActivity extends AppCompatActivity
        implements StudentListFragment.StudentListListener  {
```

```
@Override
public void itemClicked(long id) {

  //--------------------два типа утсройства

  View fragmentContainer = findViewById(R.id.fragment_container);
    if (fragmentContainer != null) {
      StudentDetailFragment  detail = new StudentDetailFragment ();
      FragmentTransaction ftr = getFragmentManager().beginTransaction();
      detail.setStudent(id);
      ftr.replace(R.id.fragment_container, detail);
      ftr.addToBackStack(null);
      ftr.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
      ftr.commit();

   } else {
      Intent intent = new Intent(this, DetailActivity.class);
      intent.putExtra(DetailActivity.EXTRA_STUDENTT_ID, (int)id);
      startActivity(intent);
       }
   }
```

9.3.15 Аргументы фрагментов

Фрагменты должны сохранять свою модульность и не должны общаться друг с другом напрямую. Если один фрагмент хочет связаться с другим, он должен сообщить об этом своему менеджеру активности, а он уже передаст другому фрагменту. Есть три основных способа общения фрагмента с активностью:

— активность может создать фрагмент и установить аргументы для него;

— активность может вызвать методы экземпляра фрагмента;

— фрагмент может реализовать интерфейс, который будет использован в активности в виде слушателя.

К каждому экземпляру фрагмента может быть прикреплен объект *Bundle*.

Создать аргументы можно следующим образом:

```
Bundle args = new Bundle();
args.putSerializable(EXTRA_MY_OBJECT, mObject);
args.putInt(EXTRA_MY_INT, mInt);
args.putCharSequence(EXTRA_MY_STRING, mString);
```

Присоединение аргументов к фрагменту должно быть выполнено после создания фрагмента, но до его добавления в активность. Фрагмент должен иметь только один пустой конструктор без аргументов. Но можно создать статический *newInstance* с аргументами через метод *setArguments*():

```
public  class StudentDetailFragment extends Fragment {
     //------------ аргументы
     private static final String ARG_STUDENT_ID = "Some String";

     public static StudentDetailFragment newInstance(UUID studId) {
         Bundle args = new Bundle();
         args.putSerializable(ARG_STUDENT_ID, studId);
         StudentDetailFragment fragment = new StudentDetailFragment();


      fragment.setArguments(args);
         return fragment;
     }
```

Доступ к аргументам можно получить в методе *onCreate*() фрагмента:

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

    UUID StudId = (UUID) getArguments().getSerializable(ARG_STUDENT_ID);
}
```

Получаем вызов *onResume*() от ОС с хостом *Activity:*

```
@Override
public void onResume() {

    super.onResume();
        Student studentb = Student.get(getActivity());
        List<Student> st = studentb.getList();
        if (adapter == null) {
            adapter = new StudentAdapter(st);
            setListAdapter(adapter);
        } else {
            adapter.notifyDataSetChanged();
        }

    }
```

О методах коммуникации между активностиями и фрагментами можно посмотреть по ссылке:
https://developer.android.com/guide/fragments/communicate

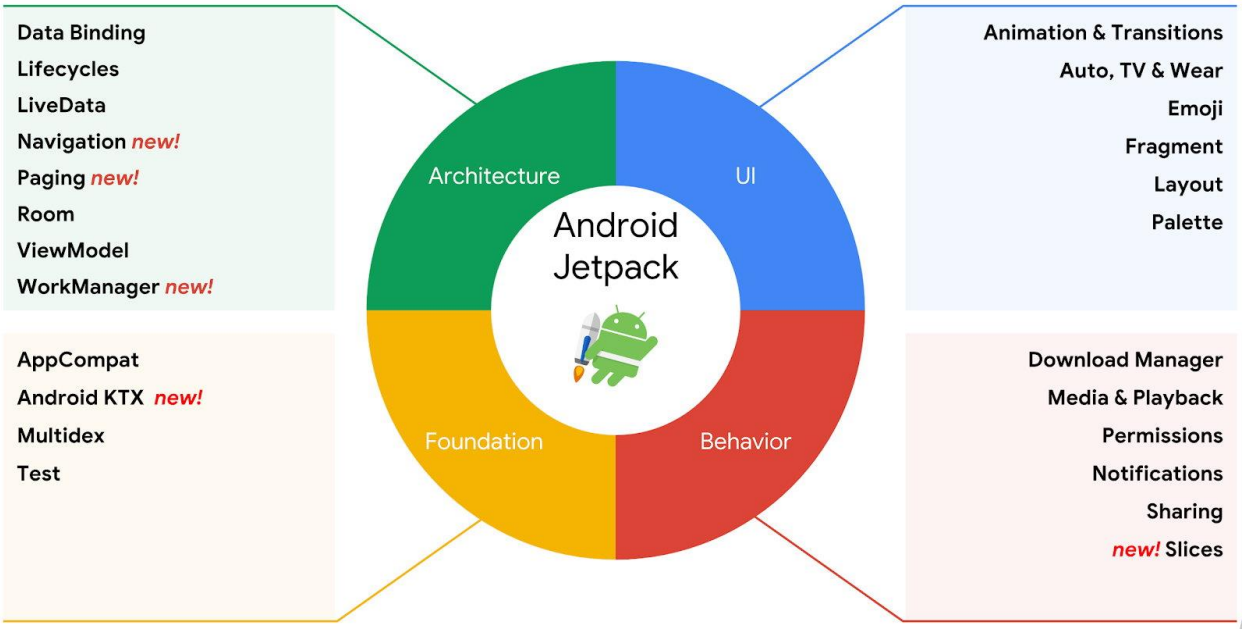
Фрагменты могут содержаться не только в активностях – они могут вкладываться в другие фрагменты.

9.4 Android Jetpack

Android Jetpack был представлен в мае 2018 г. на Google I/O. Это собранные в одном месте различные небольшие библиотеки, своего рода прокачанный наборе инструментов, который сама Google рекомендует использовать для создания быстрых, производительных и эффективных приложений.

Компоненты Android Jetpack представлены как разделяемые библиотеки, которые не являются частью базовой платформы Android. Библиотеки Android Jetpack все были перемещены в новое пространство имен **androidx. *.**

https://developer.android.com/jetpack/guide



Хотя компоненты Android Jetpack созданы для совместной работы, например, жизненного цикла и живых данных, вам не обязательно использовать их все - вы можете интегрировать часть Android Jetpack, которая решает ваши проблемы, сохраняя при этом части вашего приложения, которые уже отлично работают.

В целом инструменты достаточно стандартны, это все те же AppCompat, Android KTX, компоненты так называемой архитектуры Android: LiveData, ViewModel, Room и так далее.

WorkManager

WorkManager — это работающая на основе существующего Android библиотека, которая позволяет делать любые фоновые действия с реактивщиной в нужное время, нужной последовательности и нужных условиях и не заботиться о том, на какой версии все это будет работать:

```
WorkManager.getInstance().beginWith(firstWork)
  .then(secondWork)
  .then(thirdWork)
  .enqueue()
```

Paging

Компонент Paging 1.0.0 позволяет легко загружать и представлять большие наборы данных с быстрой и бесконечной прокруткой в RecyclerView. Он может загружать выгружаемые данные из локального хранилища, сети или и того, и другого, а также позволяет определить, как загружается ваш контент. Он работает из коробки с Room, LiveData и RxJava.

Slices

Это способ разместить пользовательский интерфейс вашего приложения внутри Google Assistant в результате поиска.

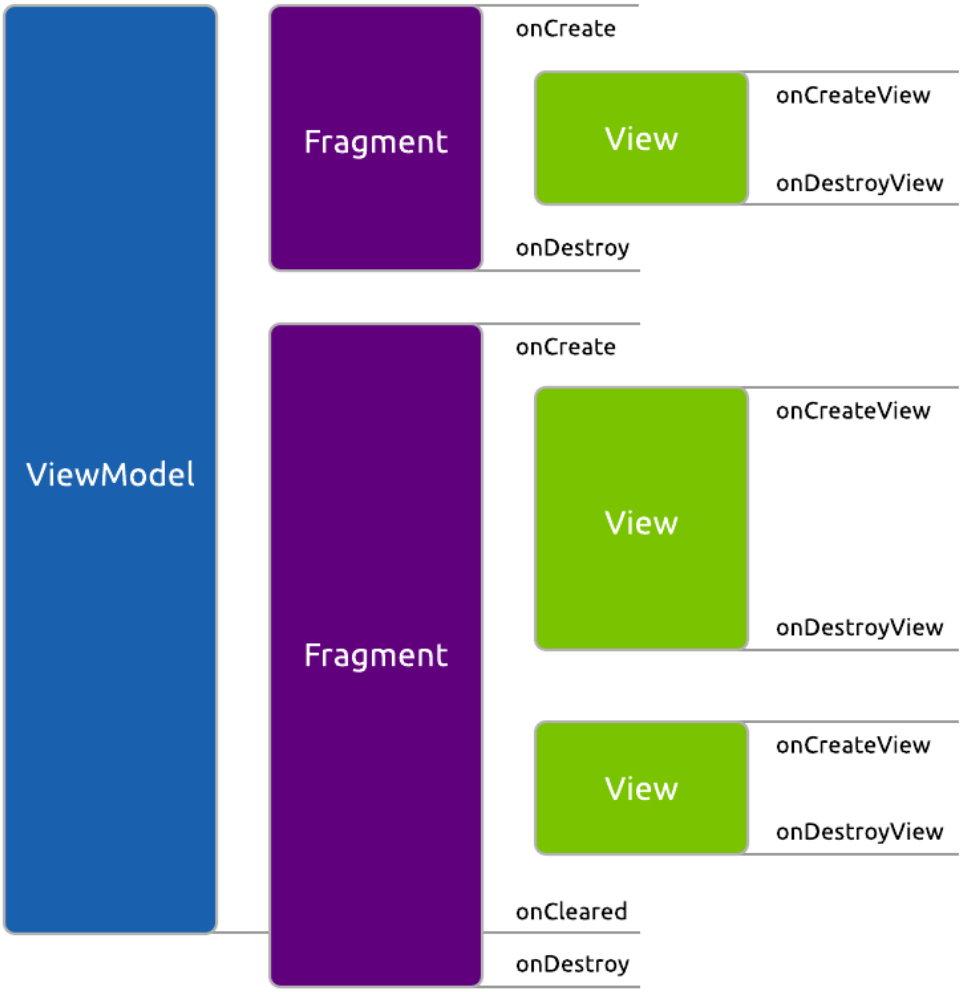
Android KTX

Использование возможностей языка Kotlin для более продуктивной работы.

9.5 Fragment в Jetpack и ViewModel

Развернутый жизненный цикл фрагмента начинается, когда Fragment впервые создается, включает все изменения конфигурации, через которые он проходит, и заканчивается, когда Fragment уничтожается навсегда, без его дальнейшего воссоздания.

В Jetpack появились новые архитектурные компоненты, в том числе ViewModel.



Jetpack **ViewModel** предназначен для решения проблемы очистки данных, восстановления фрагмента и отмены операций, запущенных во фрагменте. При воссоздании фрагмента связанный с ним экземпляр ViewModel остается таким же, как и для предыдущего экземпляра Fragment, а это означает, что данные, помещенные в ViewModel, сохранят эти изменения конфигурации, как и операции, запущенные в области ViewModel.

Жизненный цикл начинается, когда создается первый экземпляр Fragment и соответствующий экземпляр ViewModel (инициализация может быть помещена в конструктор ViewModel), и заканчивается методом *onCleared* **ViewModel**, который вызывается чуть раньше, чем вызовы *onDestroy* и *onDetach* самого последнего экземпляра Fragment.

Если вы хотите сохранить данные на уровне **ViewModel** приложения, можно использовать механизм *SavedStateHandle*.

ViewModel - рекомендуемый способ взаимодействия между фрагментами. Оба фрагмента могут получить доступ к ViewModel через содержащую их Activity. Фрагменты могут обновлять данные в ViewModel, и если данные представлены с использованием LiveData, новое состояние будет перенесено в другой фрагмент, пока он наблюдает за LiveData из ViewModel.

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

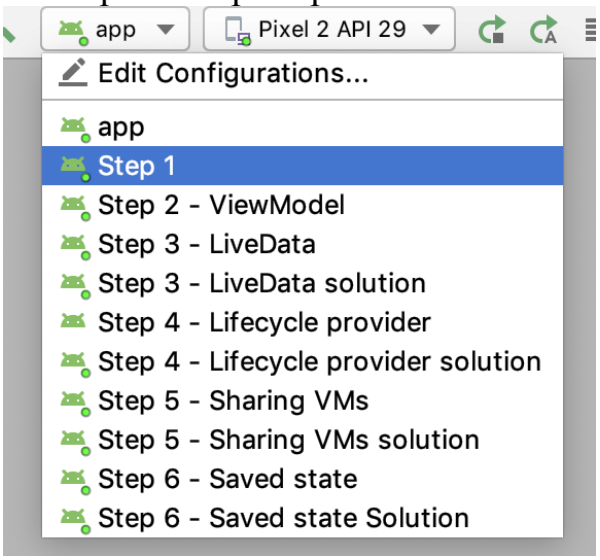
LiveData: класс хранителя данных, который можно наблюдать. Всегда хранит/кэширует последнюю версию данных. Уведомляет своих наблюдателей, когда данные были изменены.

https://developer.android.com/topic/libraries/architecture/livedata?hl=zh-tw

Activity и Fragment в Support Library, начиная с версии 26.1.0 реализуют интерфейс **LifecycleOwner**. Именно этот интерфейс и добавляет им метод **getLifecycle**, который возвращает объект **Lifecycle**. На этот объект можно подписать слушателей, которые будут получать уведомления при смене lifecycle-состояния.

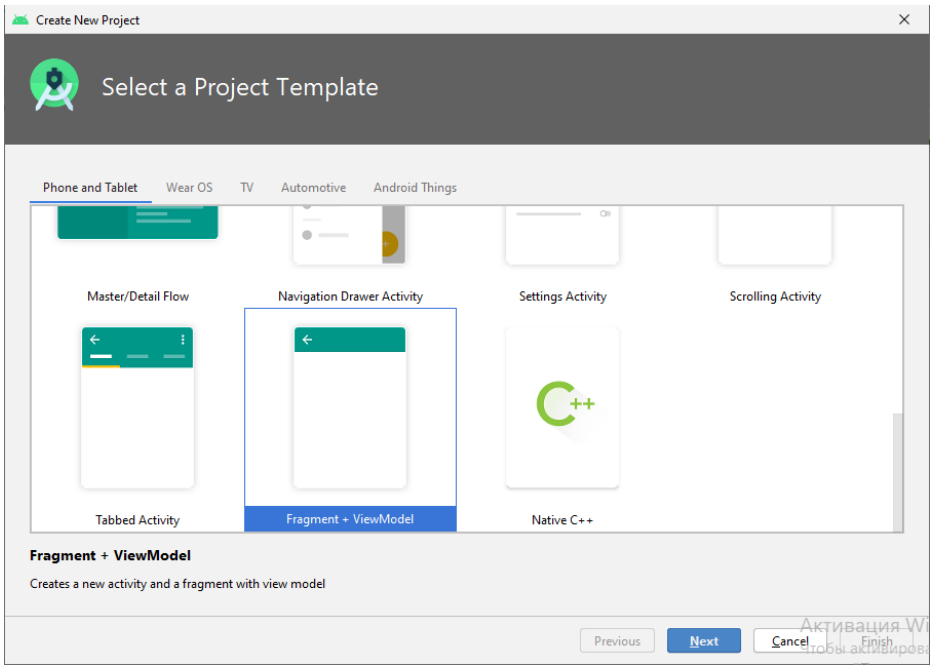
https://developer.android.com/training/basics/fragments/pass-data-between

Скачайте и разберите пример по взаимодействияю Fragment с ViewModel. Кроме того, там есть другие хорошие примеры.

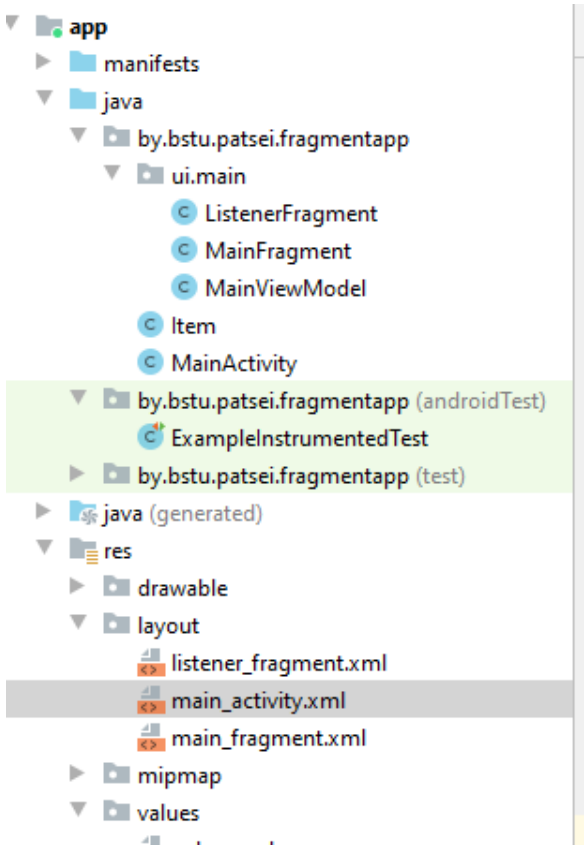


https://codelabs.developers.google.com/codelabs/android-lifecycles/index.html?index=..%2F..%2Findex#5

В Android Studio есть шаблон, который позволяет создать фрагменты с ViewModel.



Компоненты архитектуры предоставляют вспомогательный класс ViewModel, который отвечает за подготовку данных для пользовательского интерфейса. Объекты ViewModel автоматически сохраняются во время изменений конфигурации, поэтому данные, которые они хранят, немедленно становятся доступными. Например, если нужно сохранить/передать/отобразить в фрагменете какой либо объект, возложите ответственность за получение и хранение объекта на ViewModel.



Добавим классы в проект. Но для начала создадим класс Item, который будем передавать из одного фрагмента в другой.

```
@Data
public class Item {
        String data;

}
```

Храниться объект будет во ViewModel, обернутый LiveData:

```
public class MainViewModel extends ViewModel {
    // TODO: Implement the ViewModel

   private final MutableLiveData<Item> item = new MutableLiveData<Item>();

    public void setItem(Item item) {
         this.item.setValue(item);
        Log.i("TAG", "ViewModel was changed  :" + item.getData());
    }
    public MutableLiveData<Item> getItem() {
        return item;
    }
}
```

Теперь опередилим разметку для фрагметов. Первый MainFragment содержит EditText.

main_fragment.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ui.main.MainFragment">



    <EditText
        android:id="@+id/message"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="24dp"
        android:ems="20"
        android:text="MainFragment"
        android:textSize="30sp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Второй класс ListenerFragment содержит TextView

listener_fragment.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:id="@+id/main"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".ui.main.MainFragment">
```

```
<TextView
    android:id="@+id/message2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="ListenerFragment"
    android:textSize="30sp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Фрагменты хоятятся в MainFragment

main_activity.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent">

<fragment android:name="by.bstu.patsei.fragmentapp.ui.main.MainFragment"
    android:id="@+id/container"
    android:layout_weight="2"
    android:layout_height="match_parent"
    android:layout_width="match_parent" />

<fragment android:name="by.bstu.patsei.fragmentapp.ui.main.ListenerFragment"
    android:id="@+id/container2"
    android:layout_weight="1"
     android:layout_height="match_parent"
    android:layout_width= "match_parent" />

</LinearLayout>
```

MainFragment должен получить ссылку на ViewModel через ViewModelProviders по типу класса модели и передать туда данные Item. Сделать это можно в методе *onCreateView*.

```
public class MainFragment extends Fragment {
//Updater Activity
    private MainViewModel mViewModel;
    EditText editText;

    public static MainFragment newInstance() {
        return new MainFragment();
    }

    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup
container,
                             @Nullable Bundle savedInstanceState) {
        View root =  inflater.inflate(R.layout.main_fragment, container, false);

        mViewModel = ViewModelProviders.of(requireActivity()).get(MainViewModel.class);
        editText = root.findViewById(R.id.message);
        subscribeView();

        return root;
```

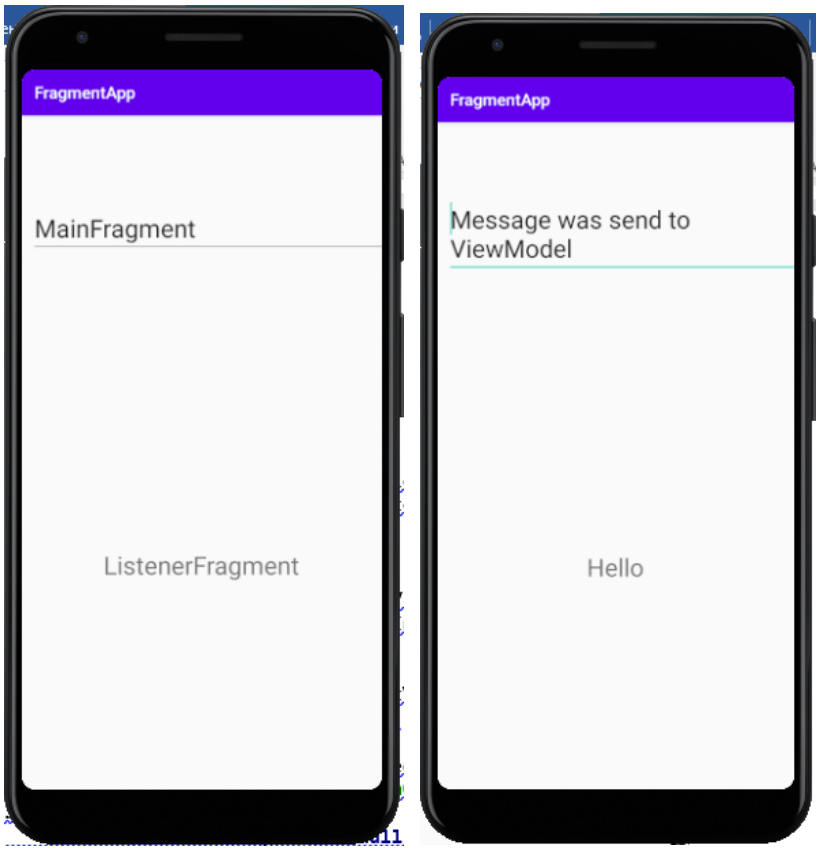
```
    }

    @Override
    public void onActivityCreated(@Nullable Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
            // TODO: Use the ViewModel
    }

    private void subscribeView() {
        Log.i("TAG", "Item  send to viewModel " );
        editText.setOnClickListener(View -> {
            Item itemNew = new Item();
            itemNew.setData(editText.getText().toString());
            mViewModel.setItem(itemNew);
        });

        mViewModel.getItem().observe(getViewLifecycleOwner(),
                new Observer<Item>() {
                    @Override
                    public void onChanged(@Nullable Item item) {
                        Log.i("TAG", "onChanged: recieved MainFragment:
"+item.getData());
                        if (item!=null) {
                            editText.setText("Message was send to ViewModel");
                        }
                    }
                }
        );

    }

}
```

ListenerFragment тоже получает ссылку на ViewModel и опредляет obsever, который срабатывает при изменении Item и заменяет текст в textView:

```
public class ListenerFragment extends Fragment {
//Listener Fragment
    private MainViewModel mViewModel;
    TextView tv2;

    public static ListenerFragment newInstance() {
        return new ListenerFragment();
    }

    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup
container,
                             @Nullable Bundle savedInstanceState) {
        View root =  inflater.inflate(R.layout.listener_fragment, container, false);
        tv2 = root.findViewById(R.id.message2);
        mViewModel = ViewModelProviders.of(requireActivity()).get(MainViewModel.class);
        observerView();
      Log.i("TAG", String.valueOf(mViewModel.getItem().hasObservers()));
     return root;
    }

    @Override
    public void onActivityCreated(@Nullable Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
    }
    @Override
```

```
    public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
    }
    private void observerView () {
        mViewModel.getItem().observe(getViewLifecycleOwner(),
                new Observer<Item>() {
                    @Override
                    public void onChanged(@Nullable Item item) {
                        Log.i("TAG", "onChanged: recieved ListenerFragment:  " +
item.getData());
                        if (item != null) {
                              tv2.setText(item.getData());
                        }
                    }
                }
        );
    }
}
```

Выполнение. Запустим приложение. Введем в edittext сообщение Hello и выполним событие click. Hello через ViewModel становиться доступноым ListenerFragment. Т.к. на observer стоит на MainFragment то и там обновляется сообщение в edittext.



По Log можно посмотреть как передавался Item.