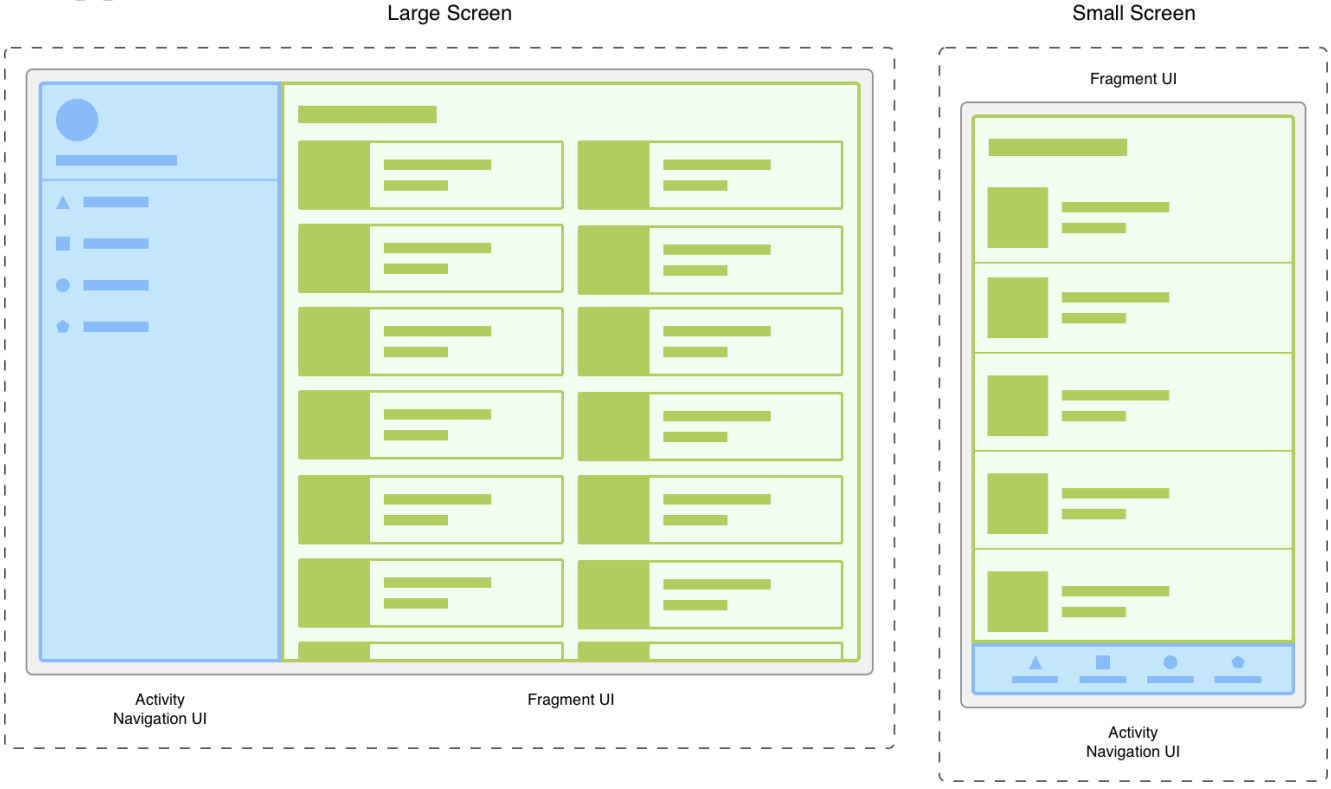
9 Фрагменты (fragments)

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.

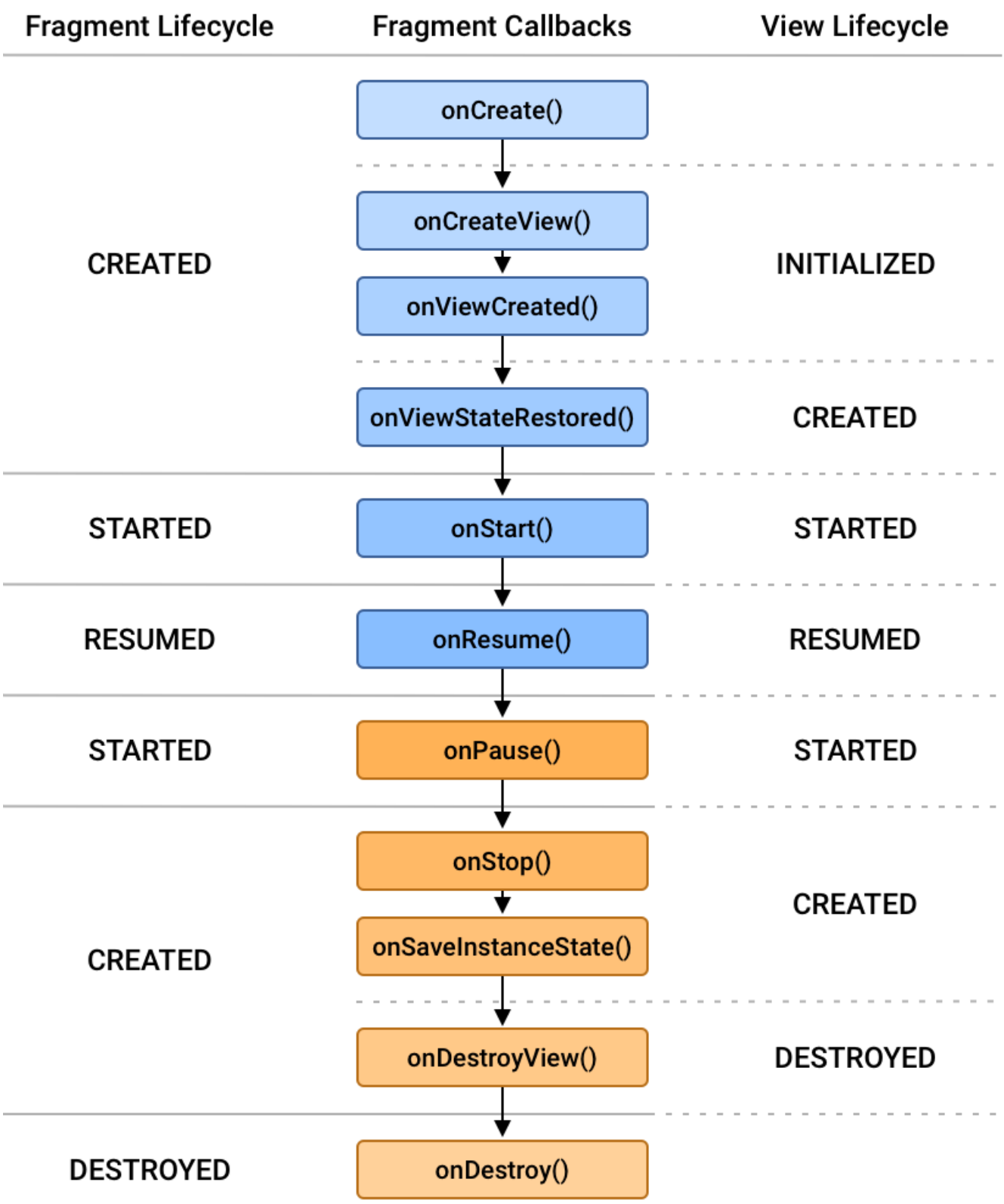
Фрагменты содержатся в активностях и находятся под их управлением.

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()

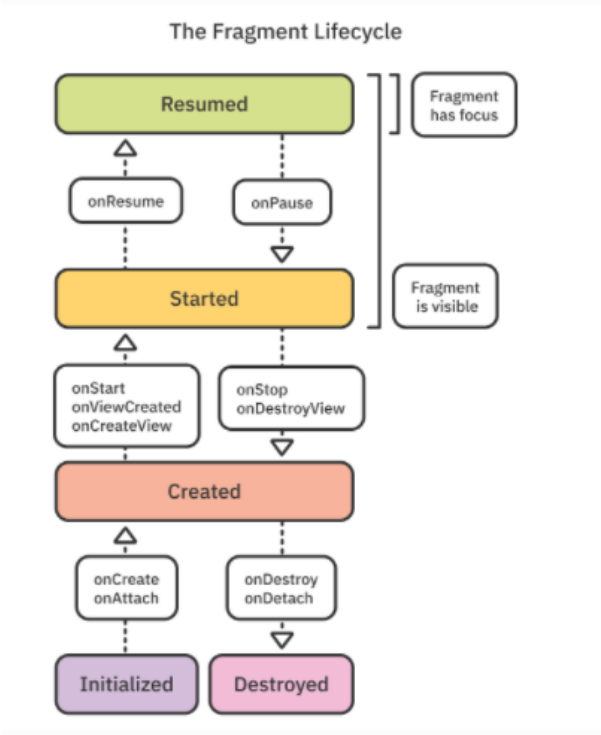
Вызывается, когда метод onCreate(), принадлежащий актиности, возвращает управление.

onDestroyView()

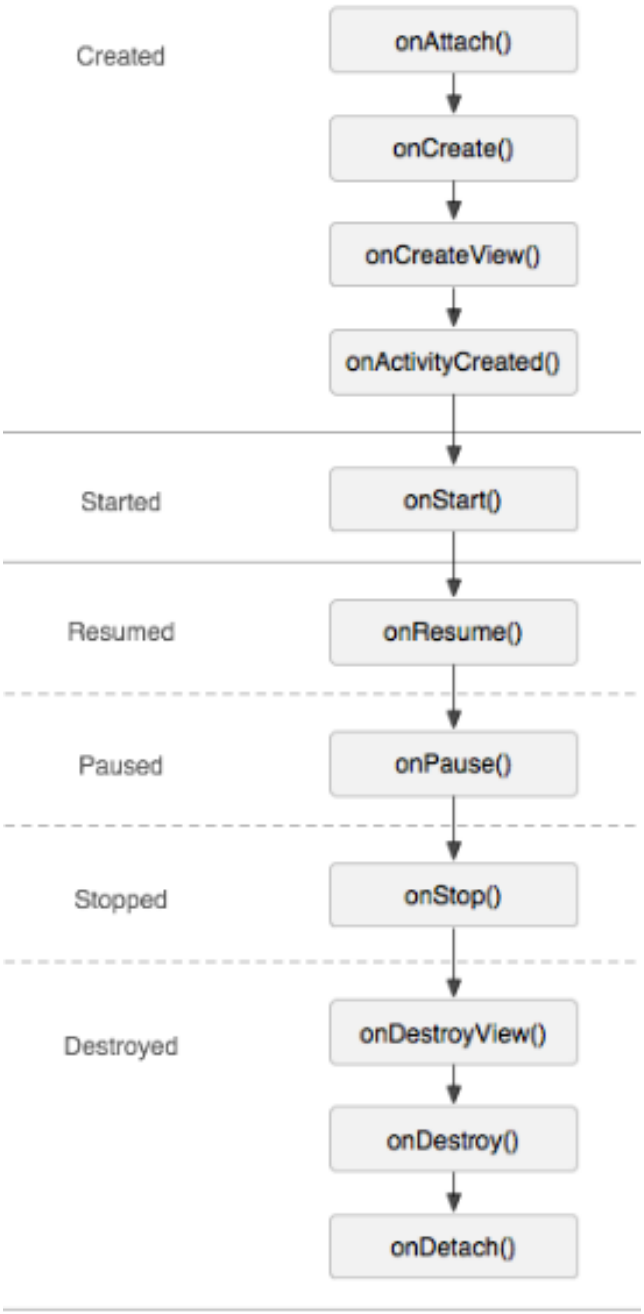
Вызывается при удалении иерархии представлений, связанной с фрагментом.

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()**.

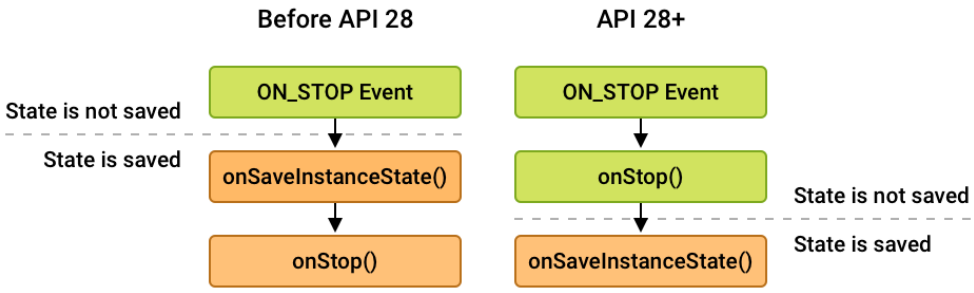


Когда активность переходит в состояние «возобновлена», можно свободно добавлять в нее фрагменты и удалять их. Таким образом, жизненный цикл фрагмента может быть независимо изменен, только пока операция остается в состоянии «возобновлена».

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().

Однако порядок обратного вызова **onStop**() и сохранения состояния с помощью **onSaveInstanceState**() различается в зависимости от уровня API. Для всех уровней API, предшествующих API 28, **onSaveInstanceState**() вызывается перед **onStop**(). Для уровней API 28 и выше порядок вызовов обратный.



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

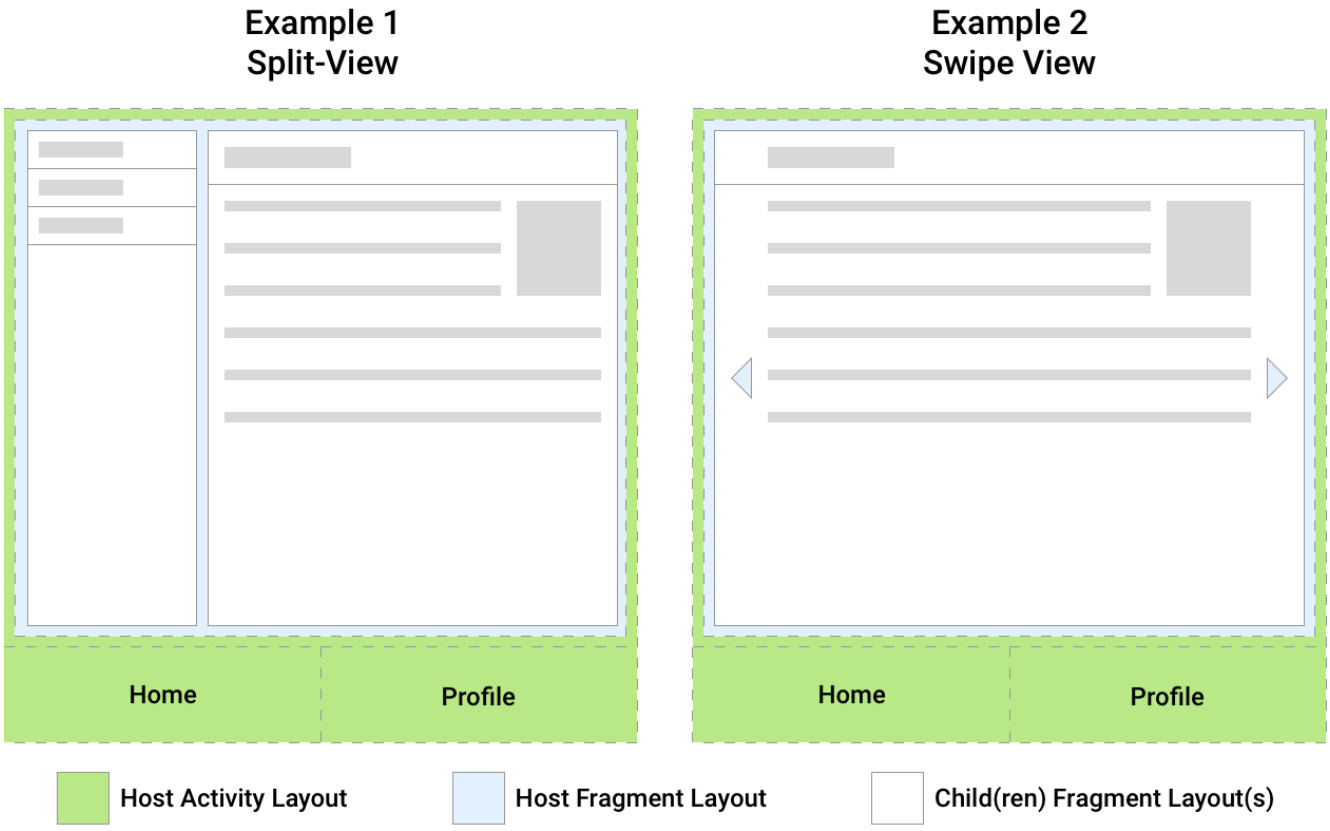
FragmentManager - класс, отвечающий действия с фрагментами добавление, удаление или замена , добавление с стек).

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

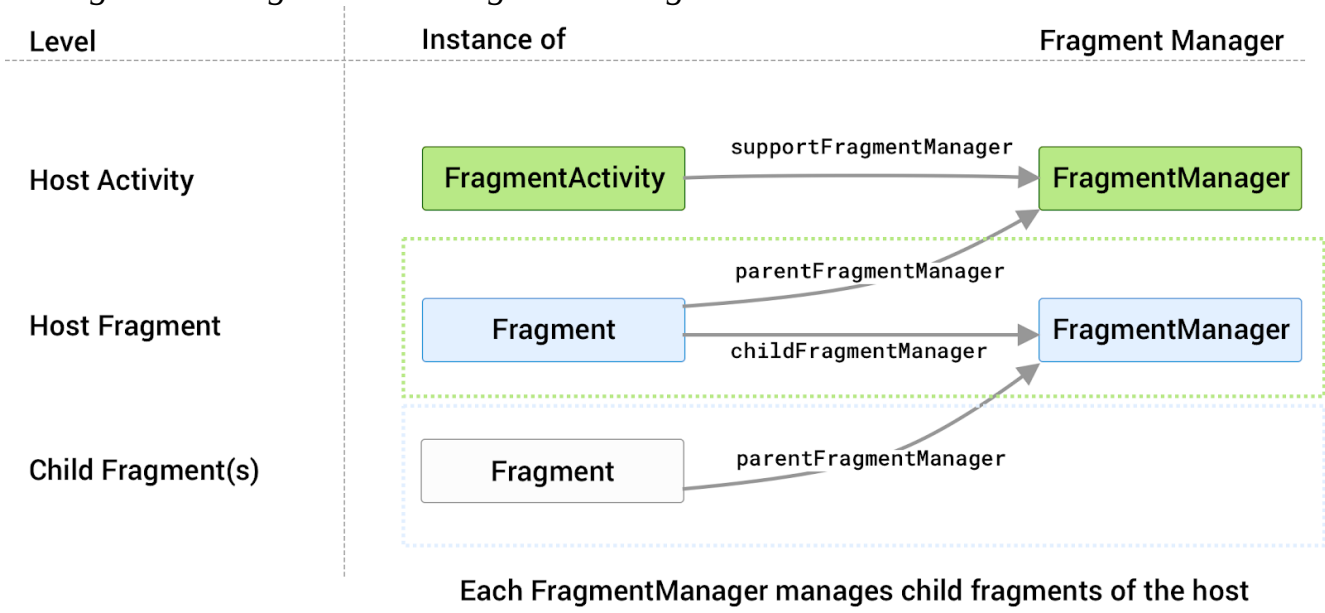
Фрагменты также могут содержать один или несколько дочерних фрагментов.



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 Сохранение состояния фрагментов

Для сохранения состояния фрагмента можно использовать:

- *Переменные*: локальные переменные во фрагменте.
- *Состояние представления*: любые данные, принадлежащие одному или нескольким представлениям во фрагменте.
- *SavedState*: данные, присущие этому экземпляру фрагмента, которые должны быть сохранены в *onSaveInstanceState*().

onSaveInstanceState(Bundle) вызывается только тогда, когда активность хоста фрагмента вызывает свой собственный *onSaveInstanceState*(Bundle).

- *NonConfig*: данные, извлеченные из внешнего источника, такого как сервер или локальный репозиторий, или данные, созданные пользователем, которые отправляются на сервер после фиксации.
  Данные NonConfig должны быть размещены за пределами вашего фрагмента, например, в ViewModel.

9.2.6 Коммуникации фрагментов

Существует 4 способа коммуникаций между фрагментами

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

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.

9.2.7 Пример приложения с фрагментами

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*.

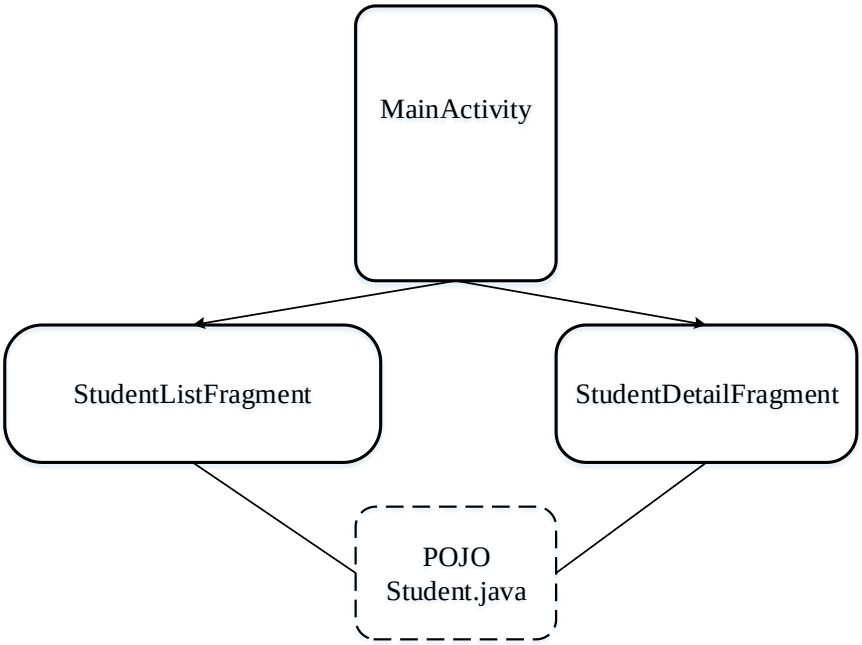


Рис. Схема приложения с фрагментами

В приложении это это может выглядеть так :

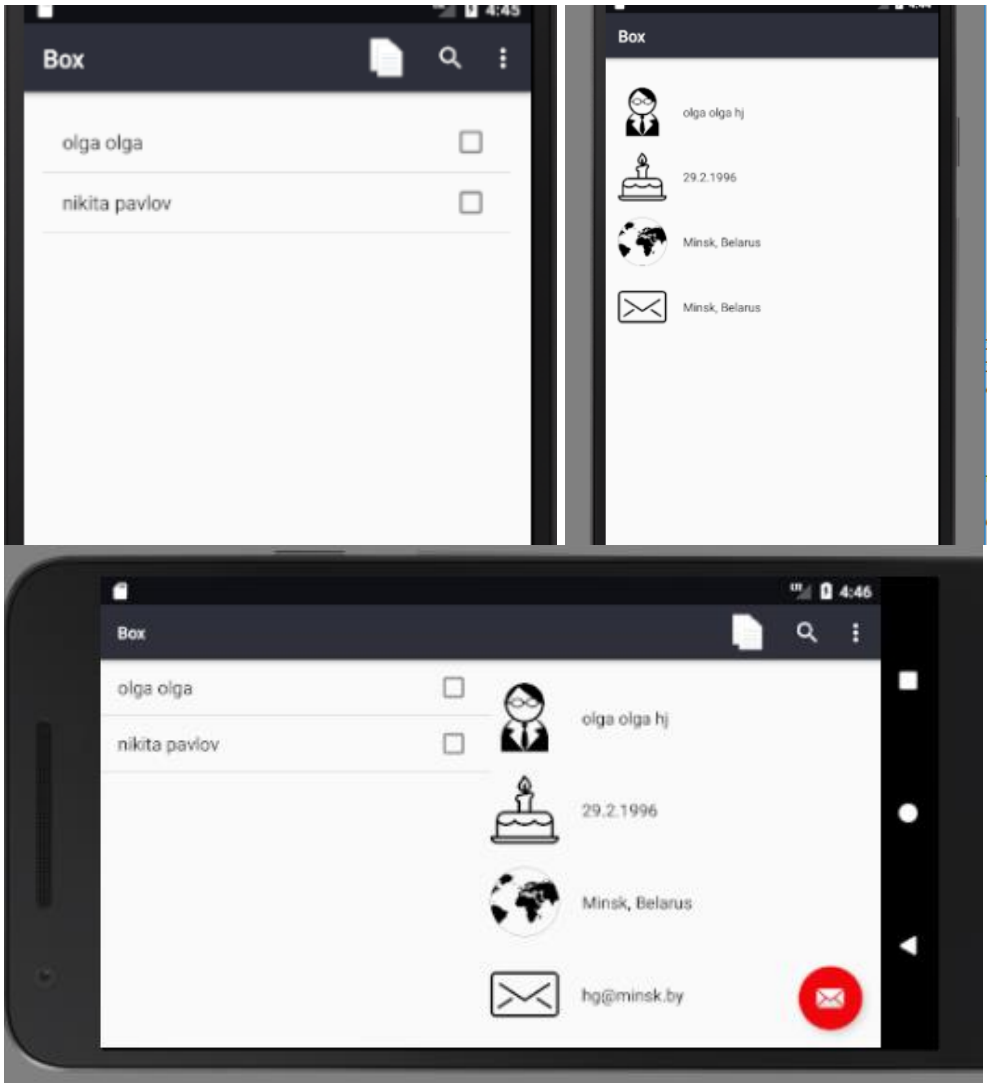


Рис. Отображение приложения с фрагментами для разной ориентации

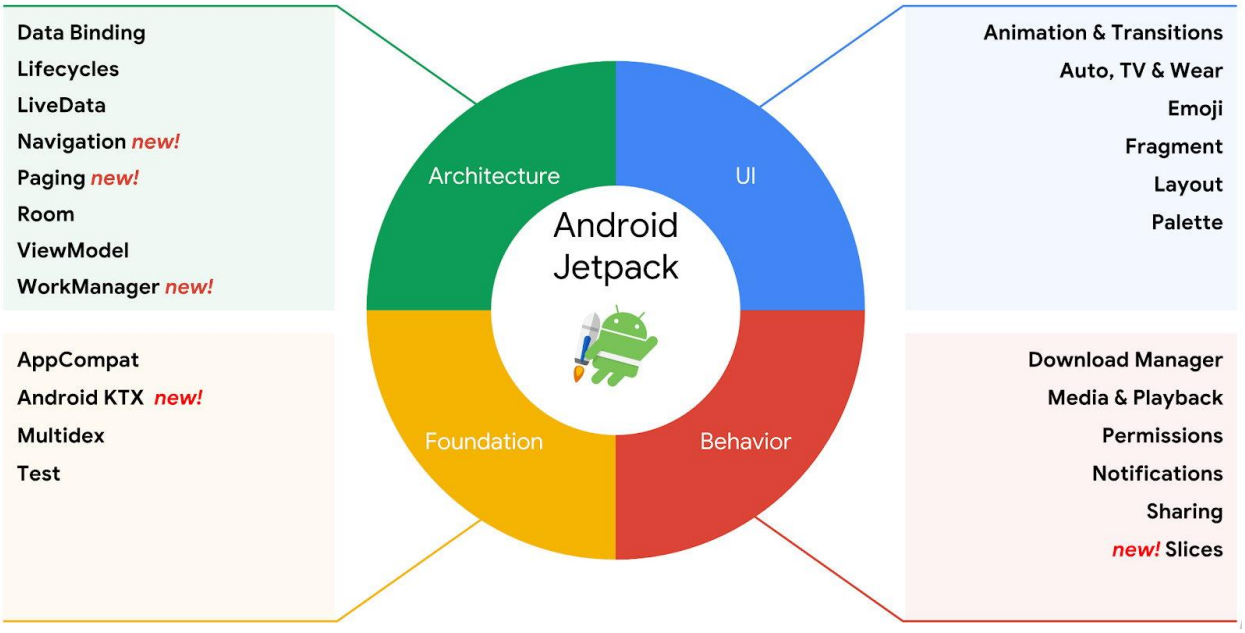
Изменим приложение так, чтобы оно по-разному выглядело и работало в зависимости от типа устройства, на котором оно выполняется. Если приложение запущено на устройстве с большим экраном, то фрагменты будут размещаться рядом друг с другом. На устройствах с малыми экранами фрагменты будут находиться в разных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

9.3 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.

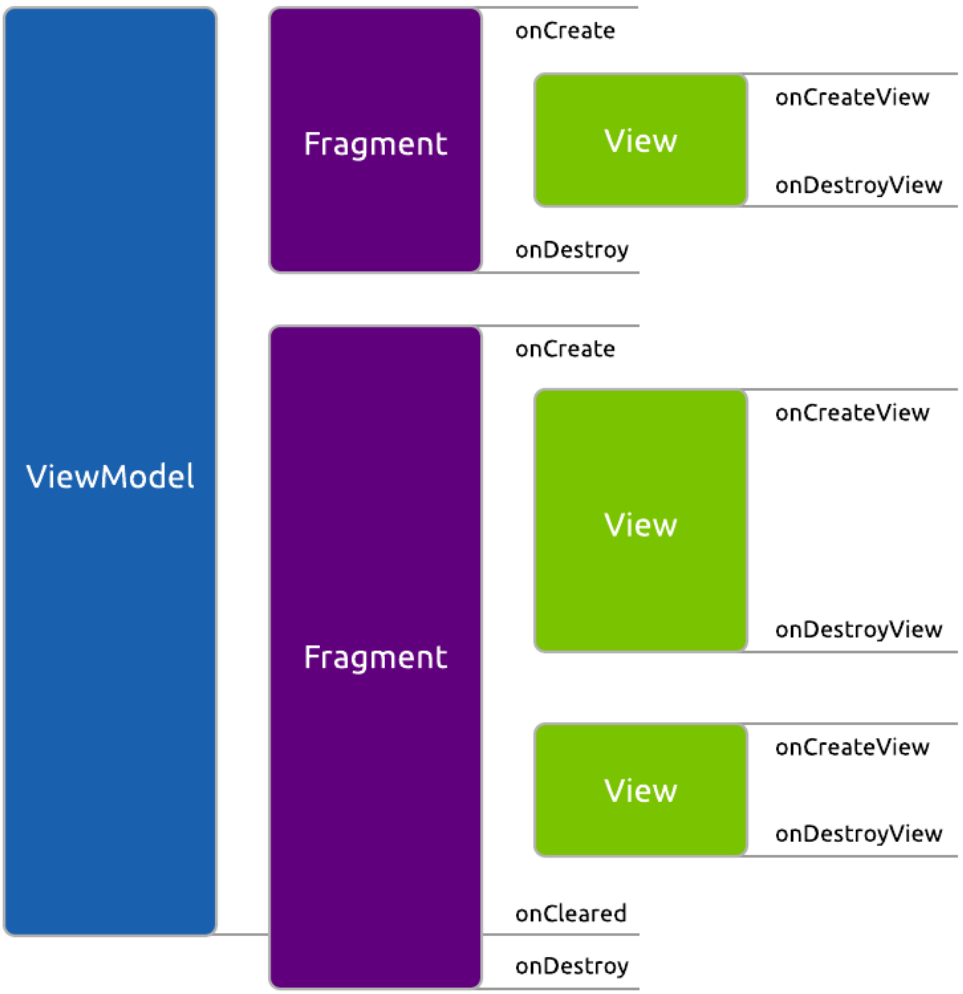
Android KTX

Использование возможностей языка Kotlin для более продуктивной работы.

9.4 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

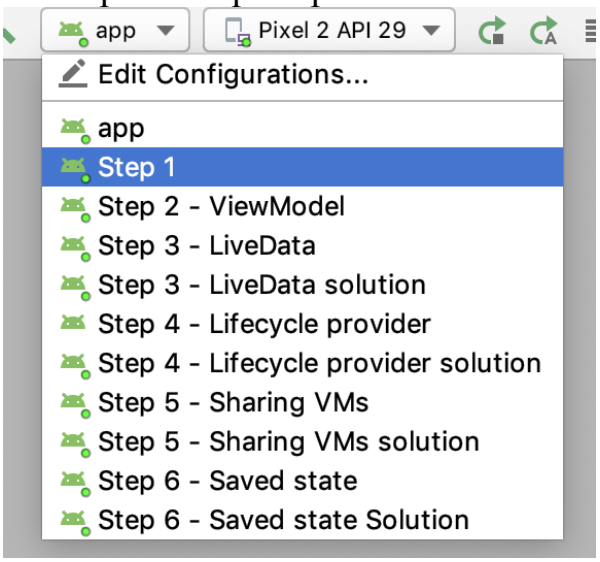
LiveData: класс хранителя данных, который можно наблюдать. Всегда хранит/кэширует последнюю версию данных. Уведомляет своих наблюдателей, когда данные были изменены.

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.

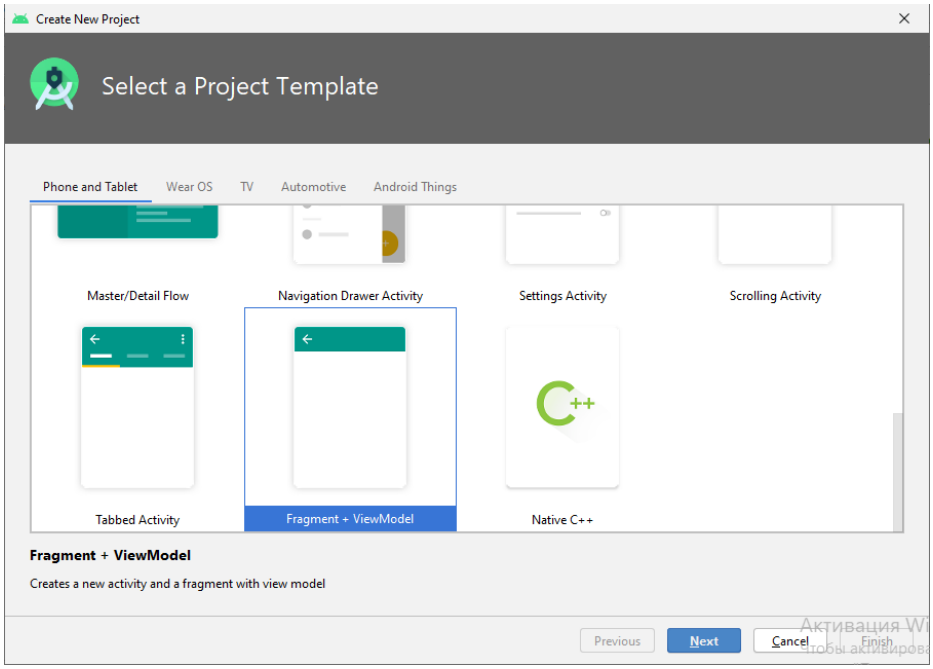
https://developer.android.com/guide/fragments/communicate

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

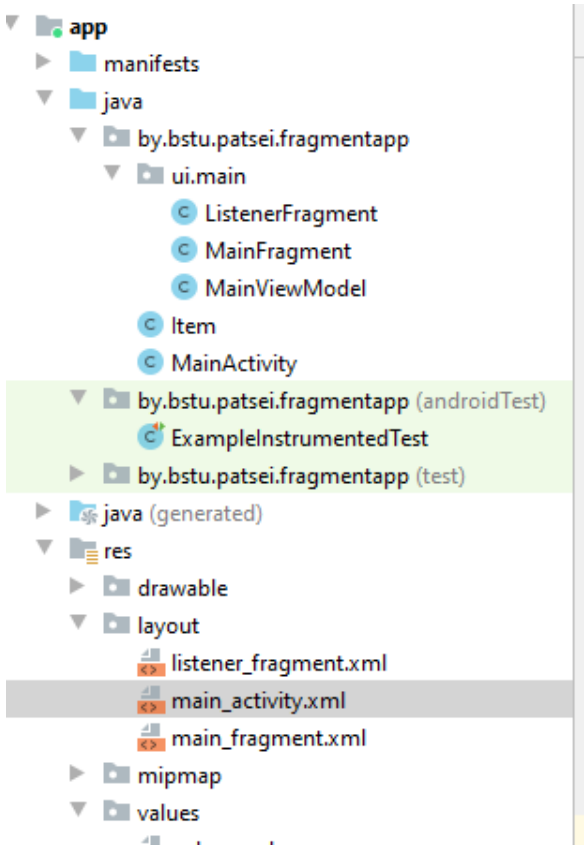


https://codelabs.developers.google.com/codelabs/android-lifecycles/index.html?index=..%2F..%2Findex#5

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 MainActivity

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());
```

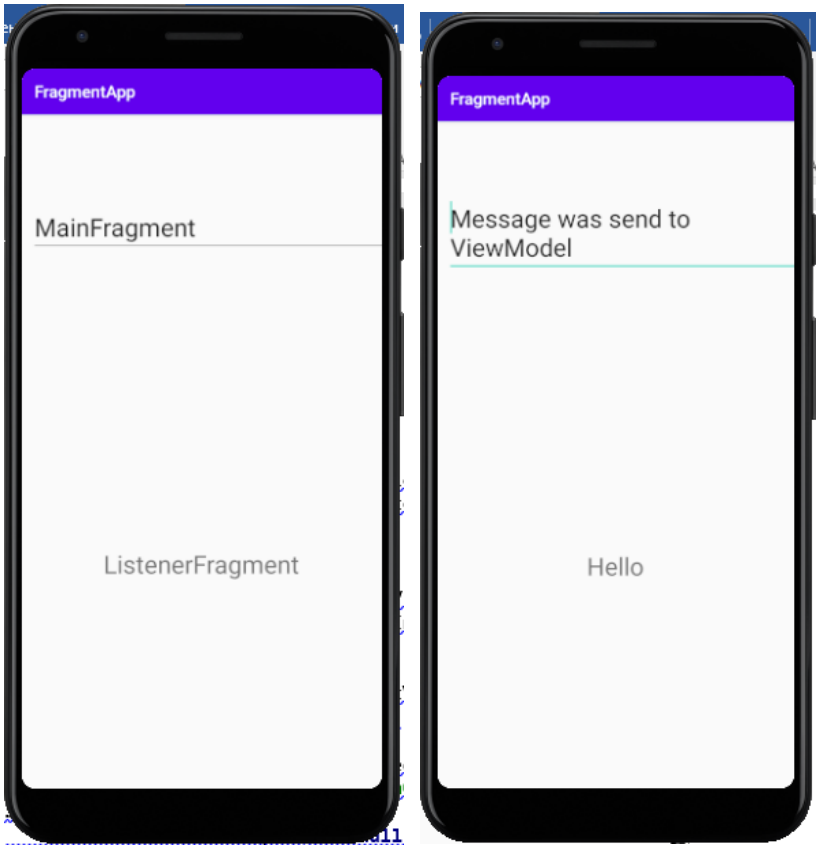
```
                        if (item!=null) {
                            editText.setText("Message was send to ViewModel");
                        }
                    }
                }
        );

    }

}
```

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

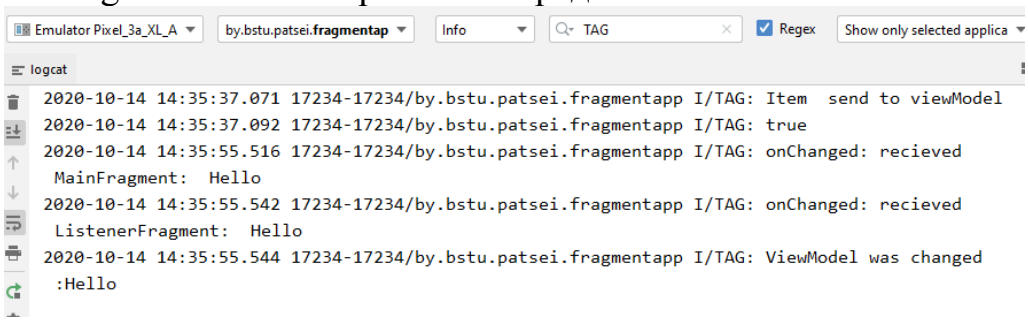
```
public class ListenerFragment extends Fragment {
//Listener Fragment
    private MainViewModel mViewModel;
    TextView tv2;

    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.



9.6 Fragment Result API

В некоторых случаях для передачи одноразового значения между двумя фрагментами или между фрагментом и его активностью можно использовать ***FragmentResultAPI***. В Fragment версии 1.3.0 и выше каждый ***FragmentManager*** реализует ***FragmentResultOwner***. Это означает, что FragmentManager может действовать как центральное хранилище результатов фрагментов.

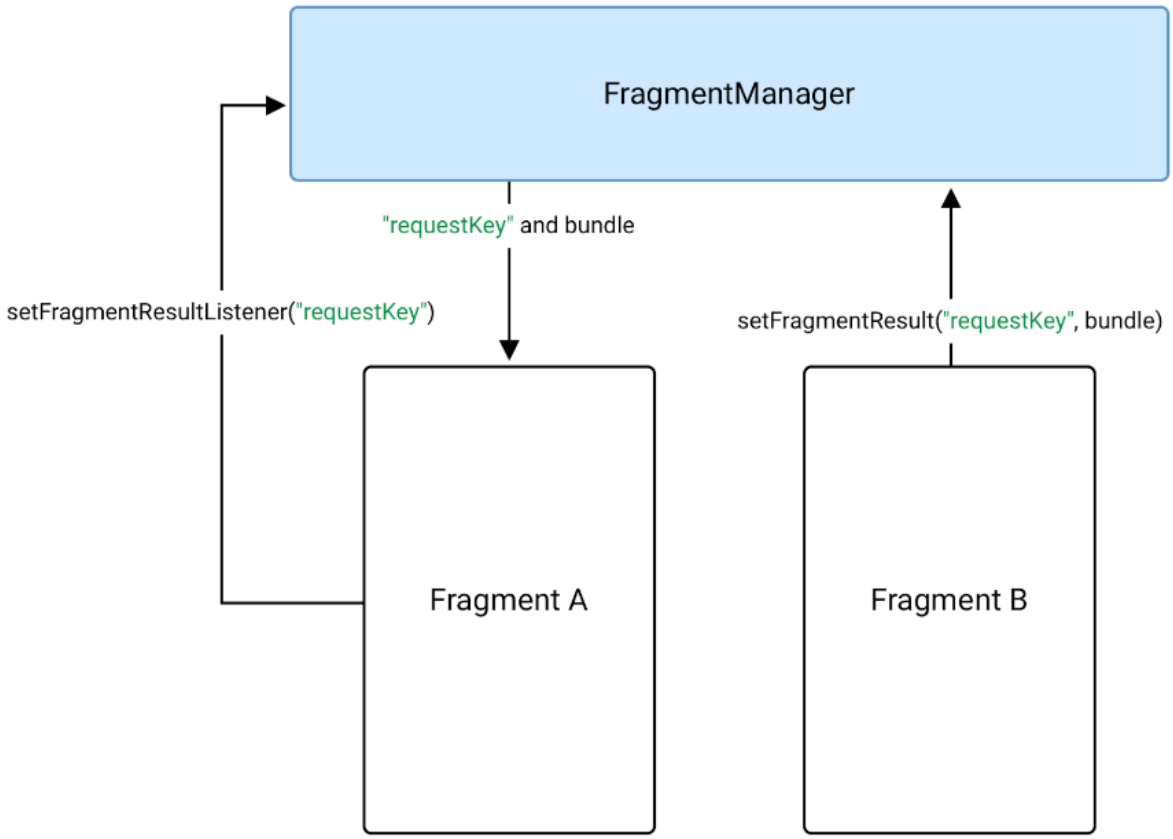
Чтобы передать данные фрагмента B в фрагмент A, устанавливается слушатель результатов на фрагмент A ***setFragmentResultListener***() для FragmentManager фрагмента A.

```
View root =  inflater.inflate(R.layout.listener_fragment, container, false);
tv2 = root.findViewById(R.id.message2);

getParentFragmentManager().setFragmentResultListener("requestKey", this, new
```

```
FragmentResultListener() {
    @Override
    public void onFragmentResult(@NonNull String requestKey, @NonNull Bundle
bundle) {
        // We use a String here, but any type that can be put in a Bundle is
supported
        String result = bundle.getString("bundleKey");
        tv2.setText(result);
    }
});
```

Во фрагменте B, производящем результат, ye;yj установить результат в том же ***FragmentManager***, используя ключ с помощью API ***setFragmentResult***().

```
View root =  inflater.inflate(R.layout.main_fragment, container, false);
editText = root.findViewById(R.id.message);
editText.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Bundle result = new Bundle();
        result.putString("bundleKey", editText.getText().toString());
        getParentFragmentManager().setFragmentResult("requestKey", result);
    }
});
```



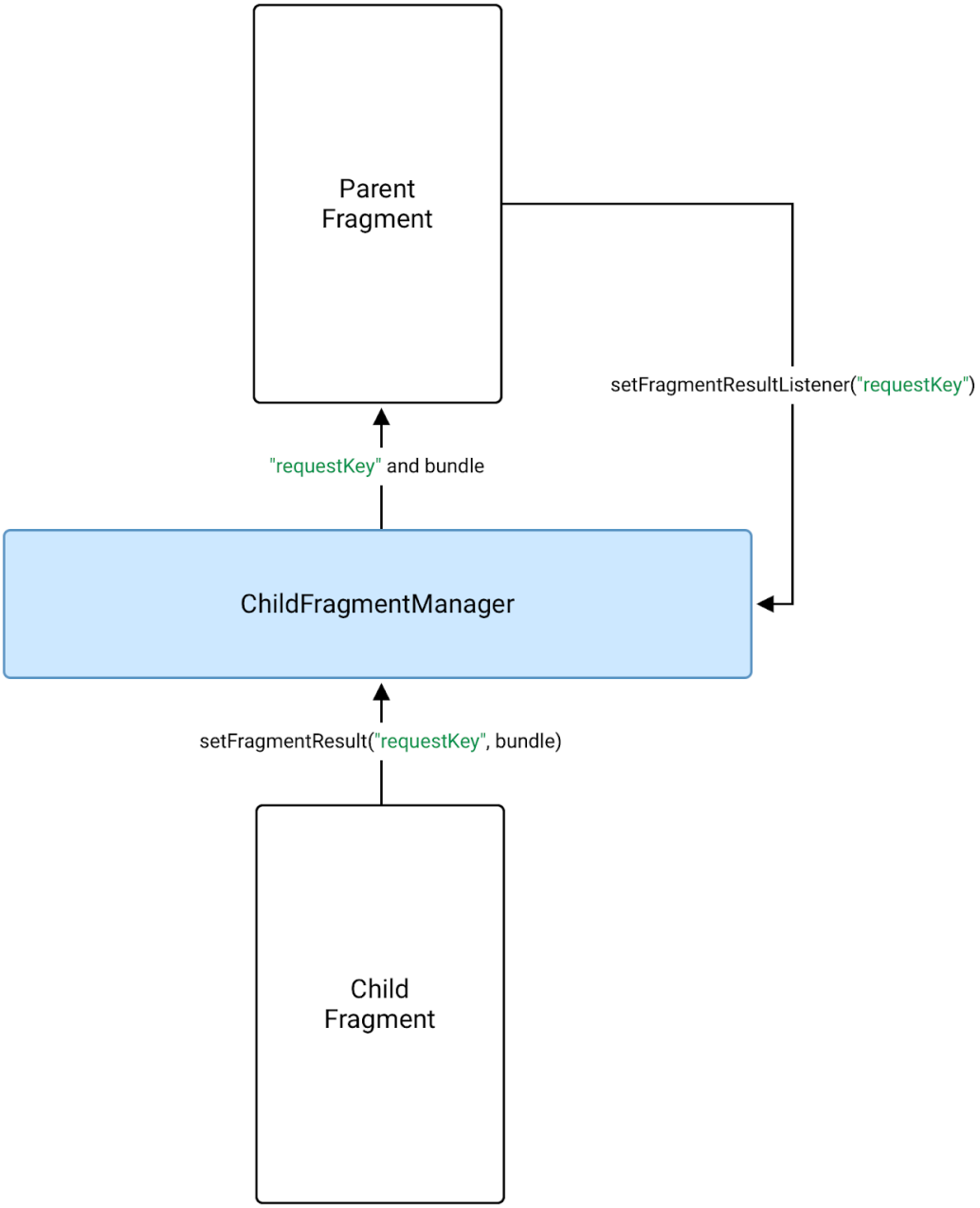
Затем фрагмент A получает результат и выполняет обратный вызов слушателя Может быть только один слушатель и результат для данного ключа. Если вызвать ***setFragmentResult***() более одного раза для одного и того же ключа, и если прослушиватель не стартован, система заменяет все ожидающие результаты обновленным результатом. Если устанавливается результат без соответствующего слушателя для его получения, результат сохраняется в ***FragmentManager*** до тех пор, пока не установите слушатель с тем же ключом. Как только результ получен

запускается обратный вызов ***onFragmentResult***(), результат очищается. Такое поведение имеет два основных последствия:

- Фрагменты в стеке не получают результатов до тех пор, пока они не будут извлечены и не будут запущены.
- когда результат установлен, обратный вызов слушателя запускается немедленно.

Поскольку результаты фрагмента хранятся на уровне ***FragmentManager***, фрагмент должен быть присоединен к вызову ***setFragmentResultListener***() или ***setFragmentResult***() с родительским FragmentManager.

Чтобы передать результат от дочернего фрагмента родительскому, родительский фрагмент должен использовать ***getChildFragmentManager***() вместо ***getParentFragmentManager***() при вызове ***setFragmentResultListener***().


Parent
Fragment
setFragmentResultListener("requestKey")
"requestKey" and bundle
ChildFragmentManager
setFragmentResult("requestKey", bundle)
Child
Fragment