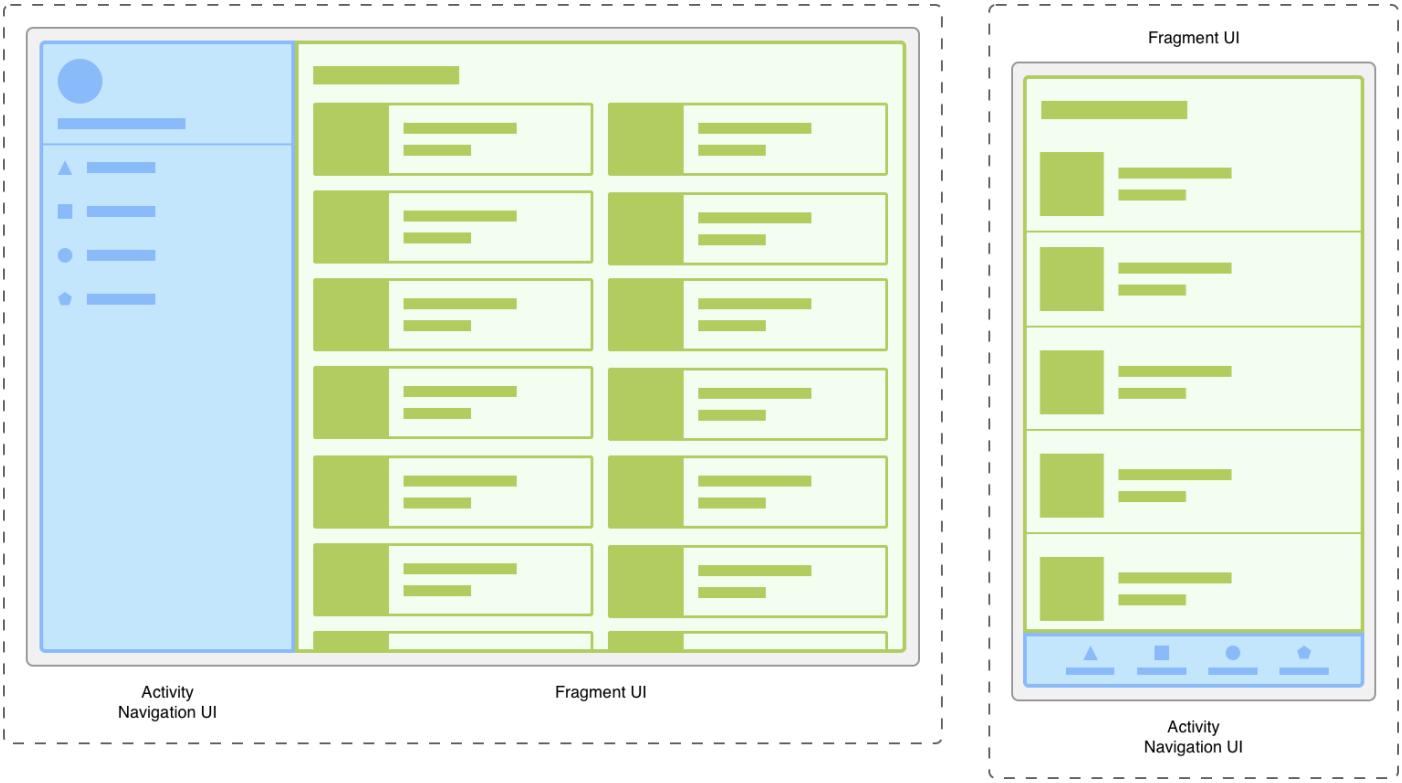
Фрагменты

Фрагмент – это повторно используемая часть пользовательского интерфейса. У него есть собственная разметка, жизненный цикл, он обрабатывает свои события. Но фрагмент не может жить сам по себе, он должен размещаться в активности или другом фрагменте.

Фрагменты обеспечивают модульность, позволяя разделить пользовательский интерфейс на логические составляющие. Например, на большом экране может отображаться полная навигационная панель и список в виде широкой сетки, а на маленьком экране – компактная панель навигации внизу и линейный список:



Динамически делать такие изменения довольно трудоёмко, удобнее отделить навигацию от контента, тогда активность будет отвечать за размещение нужных фрагментов, а сами фрагменты – отображать подходящую разметку.

Фрагменты можно добавлять, заменять или удалять прямо во время работы приложения. И все эти изменения учитываются в стеке возврата, которым управляет активность, поэтому можно возвращаться к предыдущим конфигурациям.

Несколько экземпляров одного фрагмента можно использовать в одной и той же активности, в разных активностях, или даже внутри другого фрагмента! Поэтому фрагмент должен использоваться только управления своим интерфейсом, нужно избегать зависимости одного фрагмента от другого, или управления одним фрагментом из другого.

Создание фрагмента

В среде Android Studio создавать фрагменты можно выбрав в меню File → New → Fragment → Fragment (Blank), однако шаблон такого фрагмента включает в себя дополнительный код, который не всегда нужен. Часто проще вручную создать пустой новый класс и разметку для него.

Фрагмент создаётся на базе класса `Fragment`, нужные методы переопределяются для реализации логики. В простейшем случае достаточно переопределить класс и предоставить ему разметку:

```
class MyFragment : Fragment(R.layout.my_fragment)
```

После этого можно добавить этот фрагмент в активность, либо непосредственно в XML-разметку, либо определив в разметке контейнер фрагментов, а затем программно добавить его в коде. В любом случае рекомендуется всегда размещать фрагмент в элементе `FragmentContainerView`, поскольку он обеспечивает фрагментам дополнительные удобства:

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/fragment_container_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:name="com.example.MyFragment" />
```

У `FragmentContainerView` обязательно должен быть идентификатор `id`, в противном случае программа не будет работать!

Атрибут `name` указывает какой фрагмент нужно создать. Если он отсутствует, то требуемый фрагмент нужно добавить динамически в коде программы. Для этого используется класс `FragmentManager`, который управляет всеми активностями:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    if (savedInstanceState == null) {
        supportFragmentManager
            .beginTransaction()
            .add(R.id.fragment_container, MyFragment())
            .commit()
    }
}
```

В функции `onCreate` работа с фрагментами производится только в том случае если `savedInstanceState` равен `null`, то есть активность только что запущена, а не пересоздана после изменения конфигурации – в противном случае фрагмент будет и восстановлен из `savedInstanceState`, и добавлен второй (третий, четвёртый, ...) раз.

Менеджер фрагментов

`FragmentManager` – это класс, ответственный за выполнение действий над фрагментами, такими как добавление, удаление или замена, а также добавление в стек возврата. Экземпляры `FragmentManager` не нужно создавать, система всегда готова их предоставить.

В активности получить доступ к менеджеру фрагментов можно через свойство `supportFragmentManager`.

Внутри фрагмента есть два варианта: если нужно получить менеджер фрагментов, который управляет дочерними фрагментами, используется свойство `childFragmentManager`, а если требуется доступ к родительскому менеджеру фрагментов (который управляет самим этим фрагментом), то `parentFragmentManager`.

Транзакции

При добавлении, изменении или удалении фрагментов используются транзакции – набор операций, который выполняется как единое целое. Транзакция начинается с метода `beginTransaction()` и завершается методом `commit()`, который и применяет изменения. Внутри транзакции можно выполнить другие методы, такие как `add()` или `replace()`.

```
supportFragmentManager
    .beginTransaction()
    .replace(R.id.fragment_container, MyFragment())
    .setReorderingAllowed(true)
    .addToBackStack(null)
    .commit()
```

В этом примере производится замена фрагмента (если он был) в контейнере `fragment_container` на фрагмент `MyFragment`. Метод `setReorderingAllowed(true)` оптимизирует изменения состояния фрагментов, чтобы анимация и переходы работали правильно. Вызов `addToBackStack()` добавляет транзакцию в стек возврата. Позже пользователь может отменить транзакцию и вернуть предыдущий фрагмент, нажав кнопку «Назад». Если вы добавили или удалили несколько фрагментов в рамках одной транзакции, все эти операции будут отменены при извлечении из стека возврата. В `addToBackStack()` можно указать имя, если требуется вернуться к определённой транзакции, но если это не требуется, то как в примере выше вместо имени можно указать `null`.

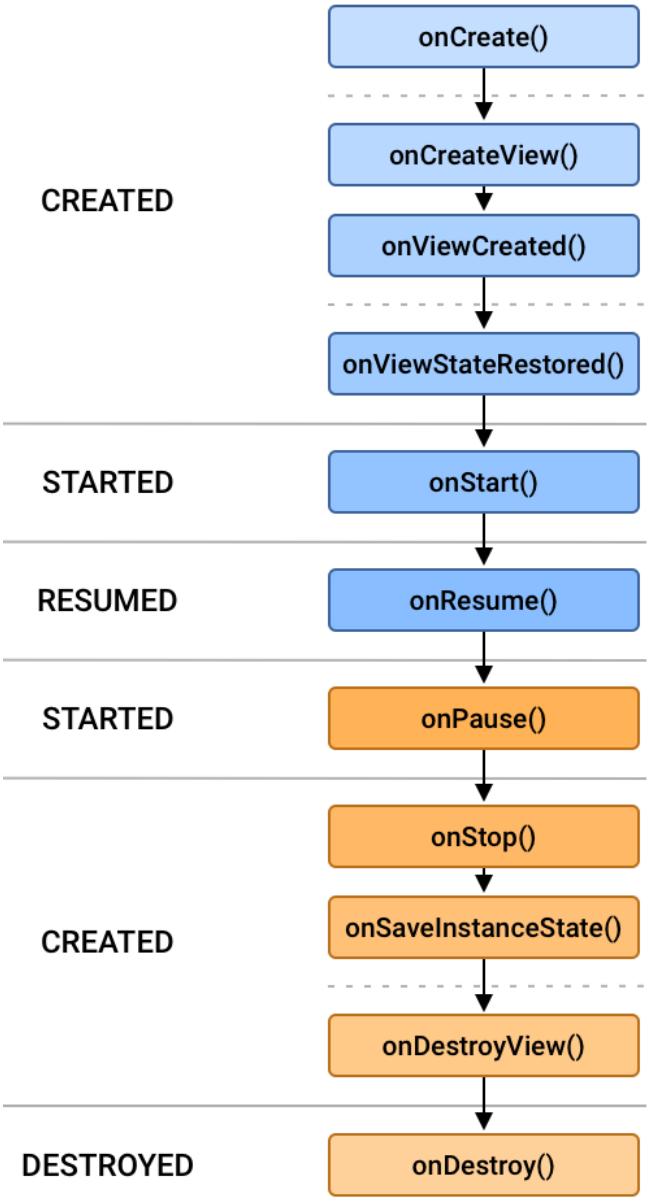
Если при выполнении транзакции не вызывается `addToBackStack()`, то удаляемый фрагмент уничтожается, и пользователь не может вернуться к нему. Если же `addToBackStack()` вызывается, то фрагмент будет просто остановлен, а при возврате пользователя его работа будет возобновлена.

Часто используемые операции во время транзакции:

- `add` – добавление фрагмента
- `remove` – удаление фрагмента
- `replace` – замена фрагмента, аналогично вызову `remove` и затем `add`
- `hide` / `show` – скрытие и отображение фрагмента

Жизненный цикл фрагмента

Как и активность, фрагмент проходит через целый ряд событий от момента своего создания, до момента разрушения:



Понимание основных этапов жизненного цикла фрагмента позволит реагировать на какие-то события, если это требуется логикой программы. В целом жизненный цикл фрагмента похож на жизненный цикл активности: они создаются, готовятся к появлению на экране, затем какое-то время находятся в рабочем состоянии, когда пользователь может с ними взаимодействовать, потом уходят с экрана, и в конце разрушаются. Но есть и отличия: разметка у активности должна быть создана и подключена в методе `onCreate`, а у фрагмента для этого есть особый метод `onCreateView`. Если же разметка создаётся самой системой, то фрагмент лишь получает уведомление об этом в методе `onViewCreated`.

У фрагмента есть ещё две функции обратного вызова, которые могут быть полезны в ряде обстоятельств:

- `onAttach` – вызывается при подключении фрагмента к активности
- `onDetach` – вызывается при отключении фрагмента от активности

Можно использовать эти методы чтобы сохранить ссылку на активность, или, наоборот, удалить её, если фрагмент отключается от активности.

Взаимодействие с фрагментом

Поскольку фрагменты могут использоваться в разных активностях и ситуациях, рекомендуется создавать их как полностью автономные компоненты, со своим поведением и разметкой. По этой причине не рекомендуется позволять фрагментам напрямую взаимодействовать с другими фрагментами или с активностью, которая их разместила. Библиотека `Fragment` предлагает два варианта связи: общий `ViewModel` и `Fragment Result API`.

Общий `ViewModel` предпочтителен если есть данные, которые должны быть общими для нескольких фрагментов, или для фрагмента и его родительской активности. Получив такой же `ViewModel`, как и активность, можно пользоваться всеми переменными и методами данного объекта.

Впрочем, если не требуется тонко отделять и разграничивать данные разных активностей, то можно использовать обычный синглтон, который предлагает язык Kotlin:

```
object CommonData {
    var mySharedData = arrayOf(...)
}
```

В данном примере создаётся объект `CommonData`, в котором есть массив `mySharedData` – и этот массив будет доступен во всём проекте.

`Fragment Result API` предполагает что фрагмент подготовил какие-то данные (например, результаты вычислений или другой обработки данных) и хочет передать их в активность. В этом случае данные помещаются в объект `Bundle`, а затем вызывается функция `setFragmentResult`:

```
setFragmentResult("myresult1", bundle)
```

Здесь `"myresult1"` – это текстовая метка результатов, она важна если фрагмент может поставлять различные типы данных, и активность должна понимать какие данные пришли в этот раз.

Активность, в свою очередь, должна подписаться на получение данных от фрагментов:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setFragmentResultListener("myresult1") { key, bundle ->
        val result = bundle.getString(...) // Чтение результатов
    }
}
```

Как только фрагмент отправляет данные с помощью функции `setFragmentResult`, они почти сразу появляются в слушателе, заданном функцией `setFragmentResultListener`.

Если требуется обеспечить более простое взаимодействие с фрагментом, вызывая его методы и читая его поля (хоть это и нарушает автономность фрагмента и рекомендации ООП), то можно и просто найти его и сохранить в какую-либо переменную:

```
val fragment = supportFragmentManager
               .findFragmentById(R.id.fragment_container) as MyFragment
```

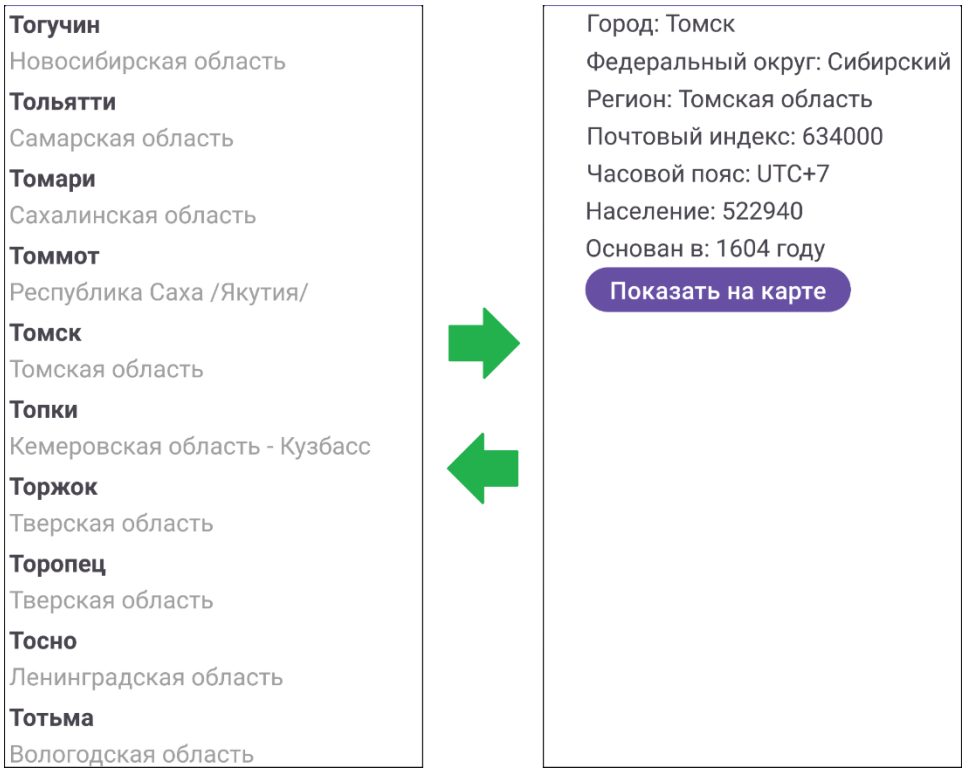
Или же присвоить создаваемому фрагменту уникальный тег, а затем найти фрагмент по этому тегу:

```
supportFragmentManager.commit {
    replace(R.id.fragment_container, "mytag")
    ...
}

val fragment = supportFragmentManager.findFragmentByTag("mytag")
```

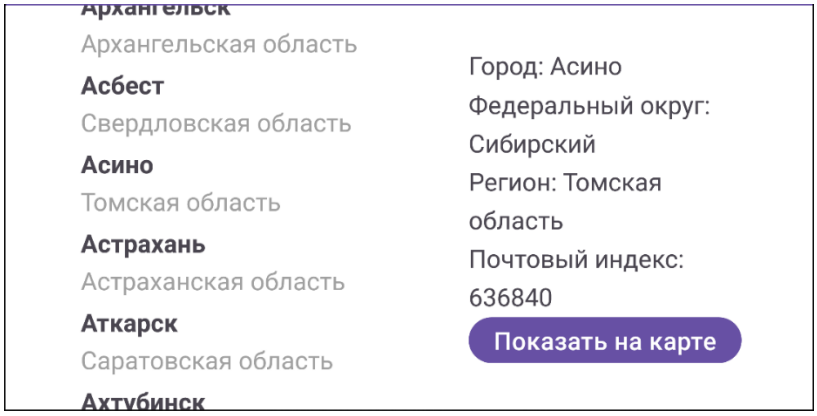
Задание

Создайте приложение, отображающее информацию о выбранном городе, а также показывающее его на карте. Да-да, как в прошлой лабораторной работе – можно даже код оттуда взять за основу! Но интерфейс на сей раз должен быть реализован через фрагменты. Если устройство находится в портретной ориентации, то весь экран занимает фрагмент со списком городов, а при нажатии на город – фрагмент со списком заменяется на фрагмент с информацией о городе:



При нажатии кнопки «Назад» происходит возврат на список городов, причем позиция прокрутки списка должна сохраняться.

При альбомной ориентации устройства на экране присутствуют сразу два фрагмента: слева со списком городов, а справа – с информацией о городе. При нажатии на название города не происходит никакой замены фрагментов, просто в правом фрагменте отображается информация о новом выбранном городе:



В данной лабораторной работе дополнительным затруднение может стать требование сохранения позиции прокрутки при возврате фрагмента со списком. Одним из вариантов решения этого затруднения является сохранение состояния списка – точнее даже не самого списка, а его менеджера разметки:

```
state = (list.layoutManager as LinearLayoutManager).onSaveInstanceState()
```

В приведённом коде у менеджера разметки вызывается метод `onSaveInstanceState`. Это немного обескураживает, ведь это функция обратного вызова, и обычно мы сами никогда её не вызываем, её всегда вызывает система! Зато после вызова этого метода (например, в событии `onPause`) в переменной `state` окажется внутреннее состояние менеджера разметки. А затем когда фрагмент со списком снова создаст свою разметку перед появлением на экране (в методе `onCreateView`) можно восстановить состояние таким образом:

```
if (state != null)
    (list.layoutManager as LinearLayoutManager).onRestoreInstanceState(state)
```

В качестве результата лабораторной работы как обычно загрузите отчёт и zip-архив с проектом.