№8 Навигация activity и списковые представления

Классификация активностей

Здесь речь пойдет не об архитектурах, а об упорядочивании и организации взаимодействия activity.

Условно можно выделить три типа активностей: *активности верхнего уровня, активности категорий и активности детализации/редактирования*.

Активности верхнего уровня представляют операции, наиболее важные для пользователя, и предоставляют простые средства для навигации. В большинстве приложений первая активность, которую видит пользователь, является активностью верхнего уровня.

Активности категорий выводят данные, принадлежащие конкретной категории, — часто в виде списка. Такие активности часто помогают пользователю перейти к активностям детализации/ редактирования. Пример активности категории — вывод списка всех товаров, групп, писем, категорий и т.д.

Активности детализации/редактирования выводят подробную информацию по конкретной записи, предоставляют пользователю возможность редактирования существующих записей или ввода новых значений. Пример активности детализации/ редактирования — активность, которая выводит подробную информацию о конкретном товаре, студенте, карте, письме и т.п.

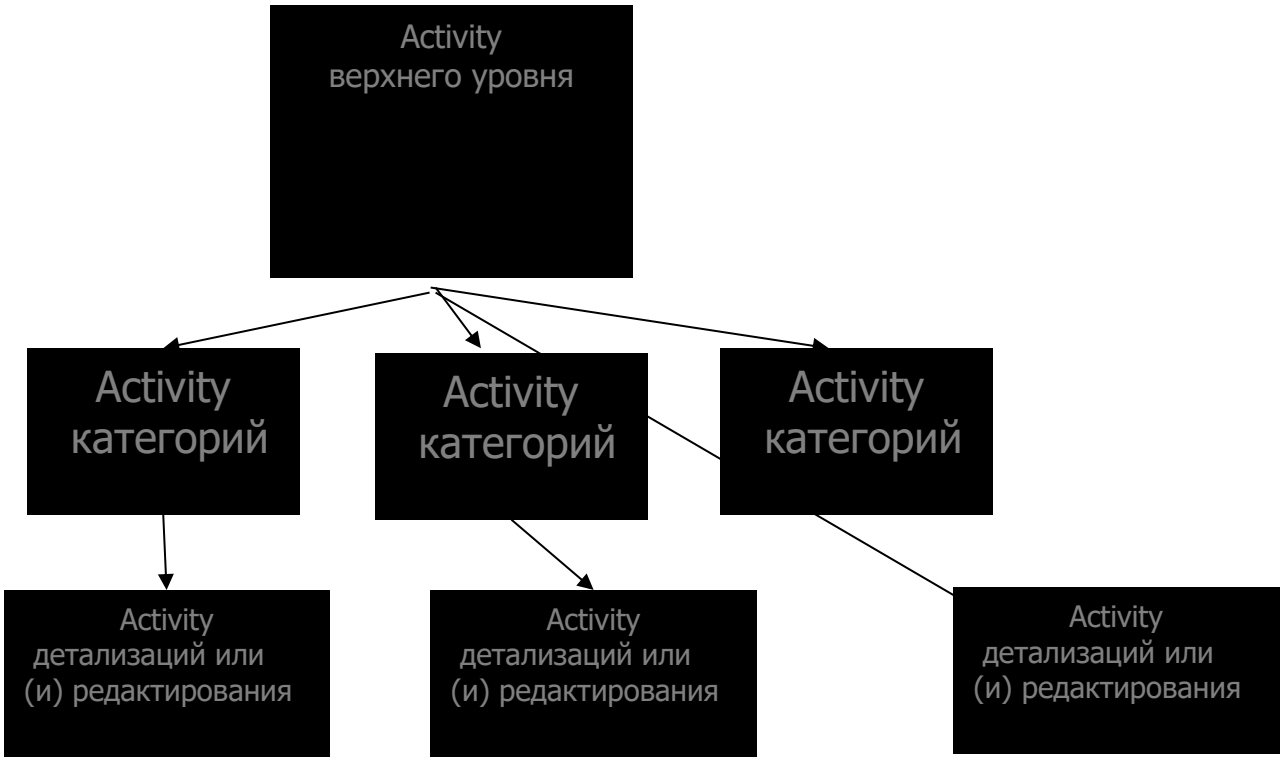
Навигация по активностям

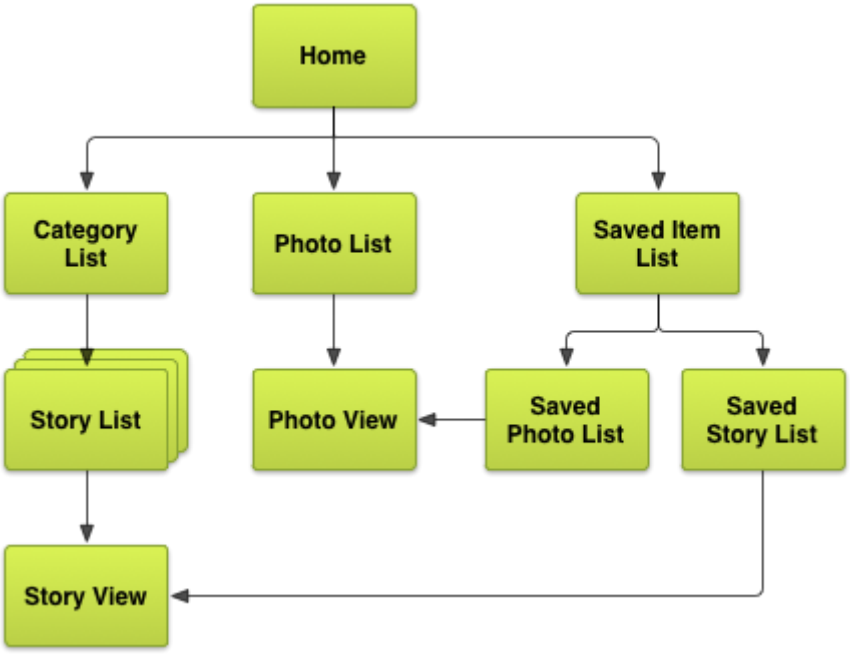
После того как вы разделите свою информацию на активности верхнего уровня, категорий и детализации/редактирования, эта классификация может использоваться для планирования навигации по приложению. Как правило, переход от активностей верхнего уровня к активностям детализации/редактирования должен осуществляться через активности категорий.

Пользователь начинает работу с приложением с активности верхнего уровня, поэтому они размещаются в начале иерархии.

Активности категорий занимают место между активностями верхнего уровня и активностями детализации/ редактирования. Пользователи будут переходить от активностей верхнего уровня к активностям категорий. В сложных приложениях иерархия может включать несколько уровней категорий и подкатегорий.

Активности детализации/ редактирования образуют нижний уровень иерархии активностей. Пользователи будут переходить к ним от активностей категорий.





Планирование нескольких размеров экрана

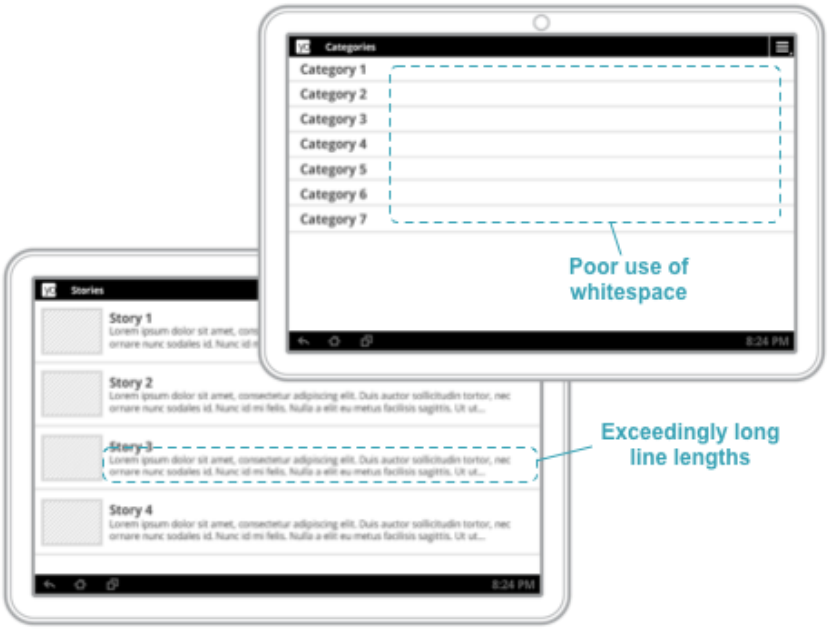
Приложениям Android необходимо адаптироваться к различным устройствам разных типов: от 3 " - 10" планшетов до 42 "телевизоров.

3-4-дюймовые экраны, как правило, подходят только для одновременного отображения одной вертикальной области содержимого, будь то список элементов или подробная информация об элементе. На таких

устройствах экраны обычно отображают один уровень в информационной иерархии (категории → список объектов → детализация объекта).

С другой стороны, более крупные экраны имеют гораздо больше доступного пространства экрана и могут представлять несколько панелей контента. В ландшафтной панели размещение происходит слева направо в порядке возрастания детализации. Пользователи привыкли к этому и в ходе использования настольных приложений и веб-сайтов.

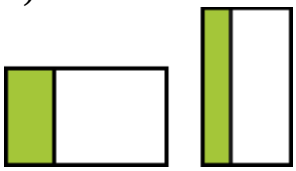
На рисунке показаны некоторые проблемы, которые могут возникнуть при крупных макетах и способы решения этих проблем с помощью многопанельных макетов:



Дизайн для нескольких ориентаций планшетов

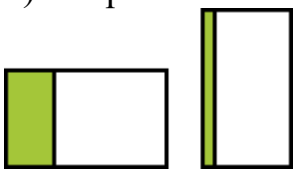
Многоуровневые макеты в ландшафтной ориентации работают достаточно хорошо из-за большого количества доступного горизонтального пространства. Однако в портретной ориентации ваше горизонтальное пространство более ограничено, поэтому вам может потребоваться создать отдельный макет для этой ориентации.

1)Растягивание

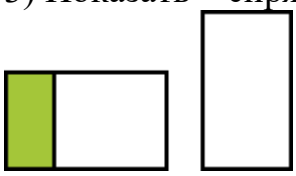


Самая простая стратегия состоит в том, чтобы просто растянуть ширину каждой панели, чтобы наилучшим образом представить содержимое. Панели могут иметь фиксированную ширину или принимать определенный процент от доступной ширины экрана.

2) Сворачивание – разворачивание

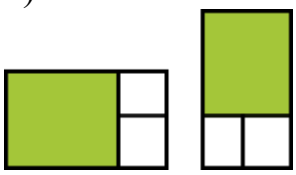


3) Показать – спрятать

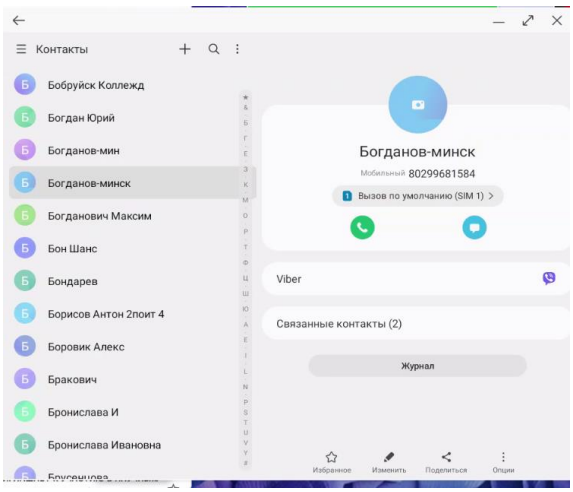


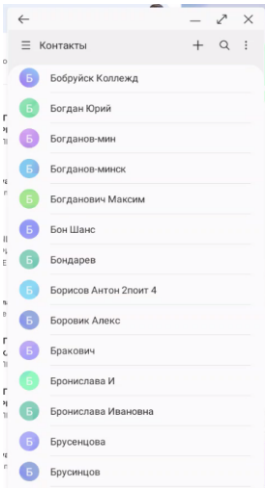
В этом случае левая панель полностью скрыта в портретном режиме. Тем не менее, левая панель должна быть доступна через экранное изображение (например, кнопку). Обычно рекомендуется использовать кнопку «Вверх» в панели действий.

4) стек



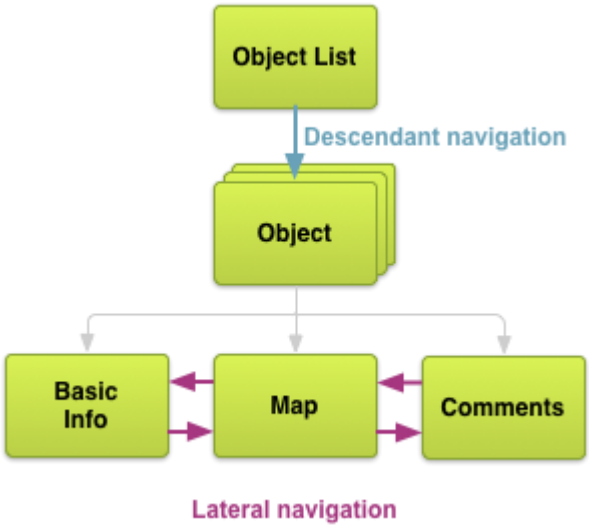
Последняя стратегия состоит в том, чтобы вертикально складывать горизонтально расположенные панели в портрете.



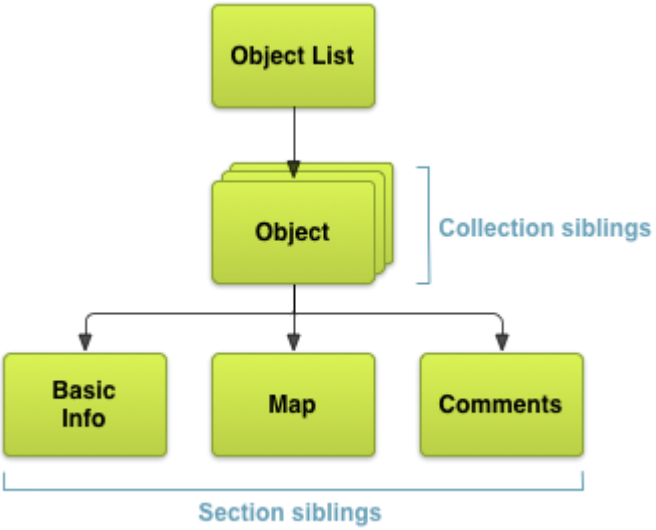


Горизонтальная и вертикальная навигация

Помиомо иерархической организации активностей, позволяющей пользователям спускаться вниз по иерархии экранов, существует горизонтальная (боковая) навигация, позволяющая пользователям получать доступ к экранам сиблинга.



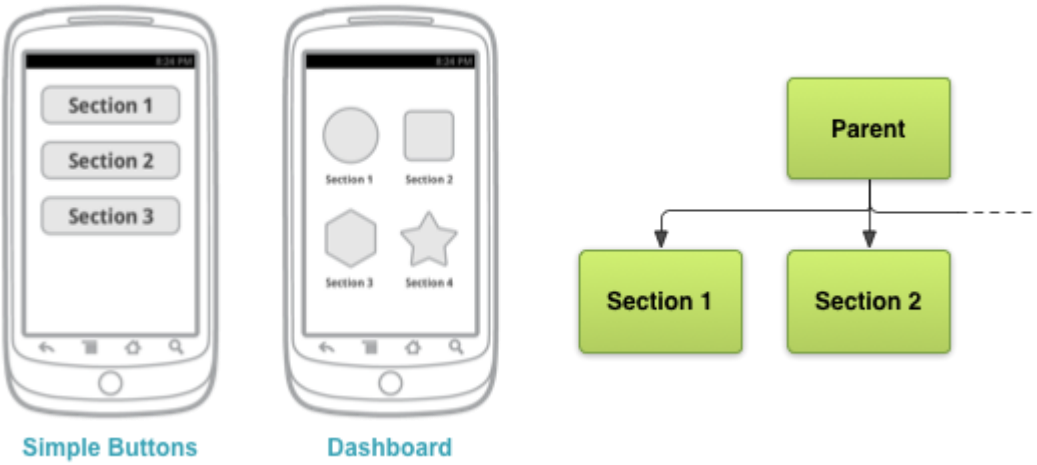
Существует два типа активнойстей для сиблинга: связанные с коллекцией и связанные с разделом.

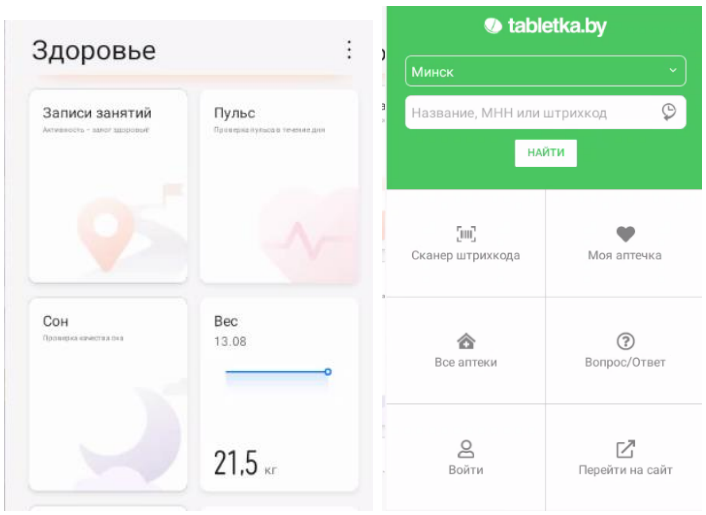


Секционные экраны представляют разные разделы информации о родителе. Например, один раздел может отображать текстовую информацию об объекте, а другой может предоставлять карту географического местоположения объекта. Количество экранов, связанных с разделом для данного родителя, обычно невелико.

Горизонтальная и вертикальная навигации может предоставляться с использованием списков, вкладок и других шаблонов пользовательского интерфейса.

Навигация на основе кнопок

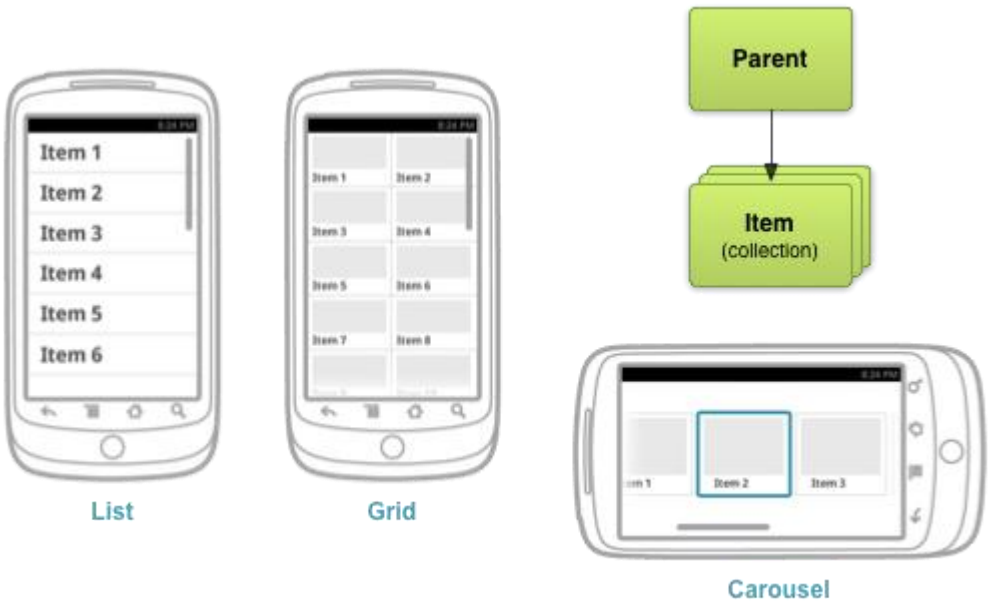




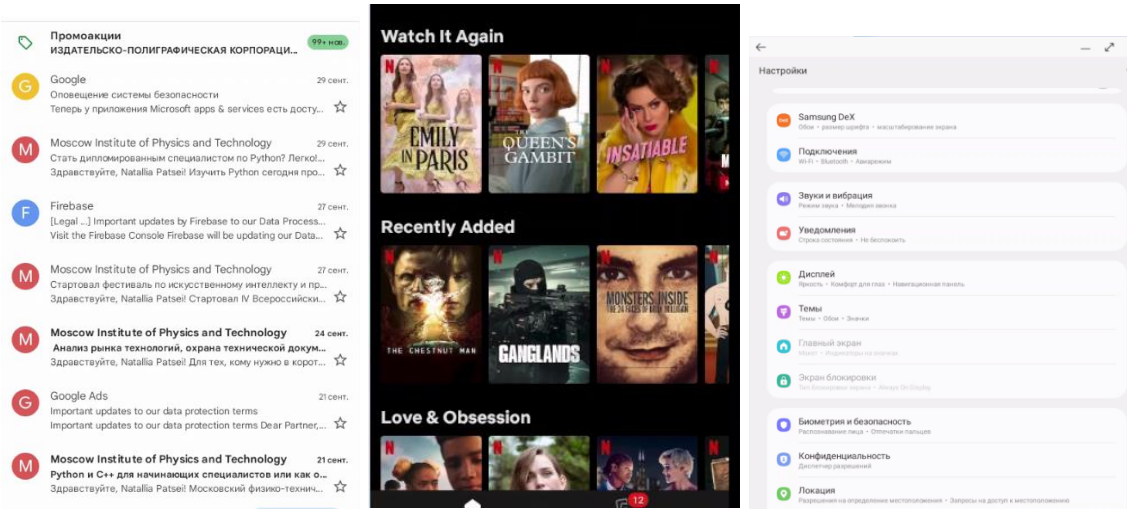
Простым шаблом для доступа к различным разделам приложений верхнего уровня является шаблон на основе кнопок или панелей. Этот шаблон - хороший способ представить все разделы приложения.

Списки, сетки и стеки

Для экранов с коллекциями, и особенно с текстовой информацией, списки с вертикальной прокруткой часто являются наиболее простым и знакомым видом интерфейса. Еще можно использовать фотографии или видео, вертикально прокручивающиеся сетки элементов, списки горизонтальной прокрутки или стеки (называемые картами). Эти элементы пользовательского интерфейса обычно лучше всего подходят для представления коллекций элементов или больших наборов дочерних экранов.

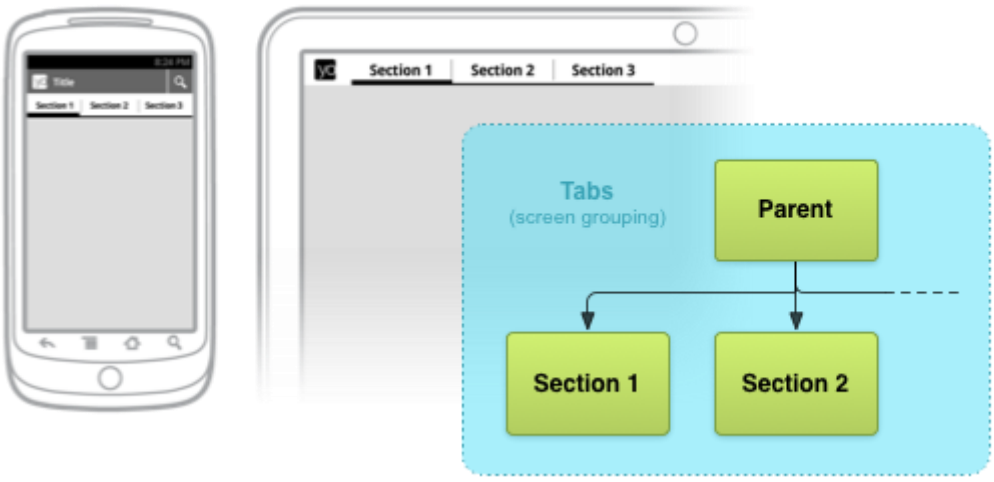


В этом шаблоне есть несколько проблем. Глубокая навигационная система на основе списков приводит к большему количеству касаний, необходимых для доступа к части контента, что приводит к плохому пользовательскому опыту.

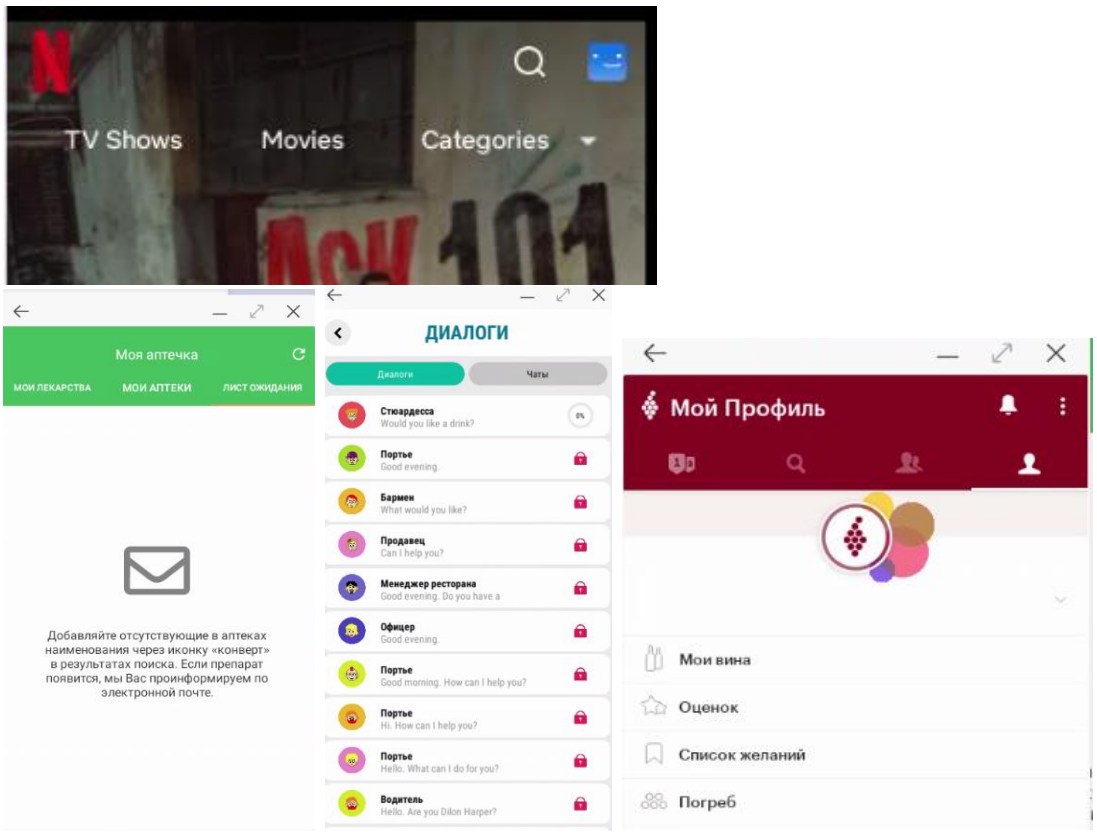


Tabs

Использование вкладок - популярное решение для горизонтальной навигации. Вкладки больше подходят для небольших наборов (4 или меньше) экранов, связанных с разделом.



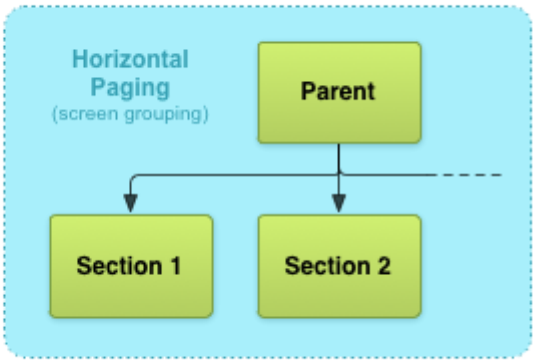
При использовании вкладок применяются несколько рекомендаций. Вкладки должны быть постоянными. При выборе вкладки индикаторы вкладок должны оставаться доступными в любое время. Переключатели вкладок не должны рассматриваться как история. Например, если пользователь переключается с вкладки A на вкладку B, нажатие кнопки «Назад» не следует переустанавливать вкладку A. Наконец, и, самое главное, вкладки чаще должны отображаться в верхней части экрана.



Swipe Views

Другой популярной навигацией является горизонтальная прокрутка.

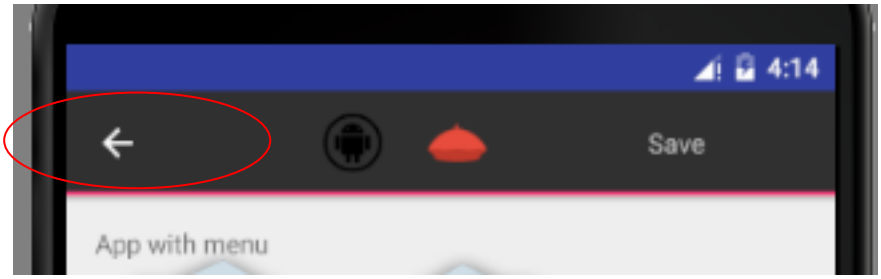
Использование вертикальных списков также привести к неудобным взаимодействиям пользователей и плохому использованию места на больших экранах, поскольку элементы списка обычно охватывают всю ширину экрана, но имеют фиксированную высоту. Один из способов - предоставить дополнительную информацию в отдельной горизонтальной панели, примыкающей к списку.



Навигация вперед- назад

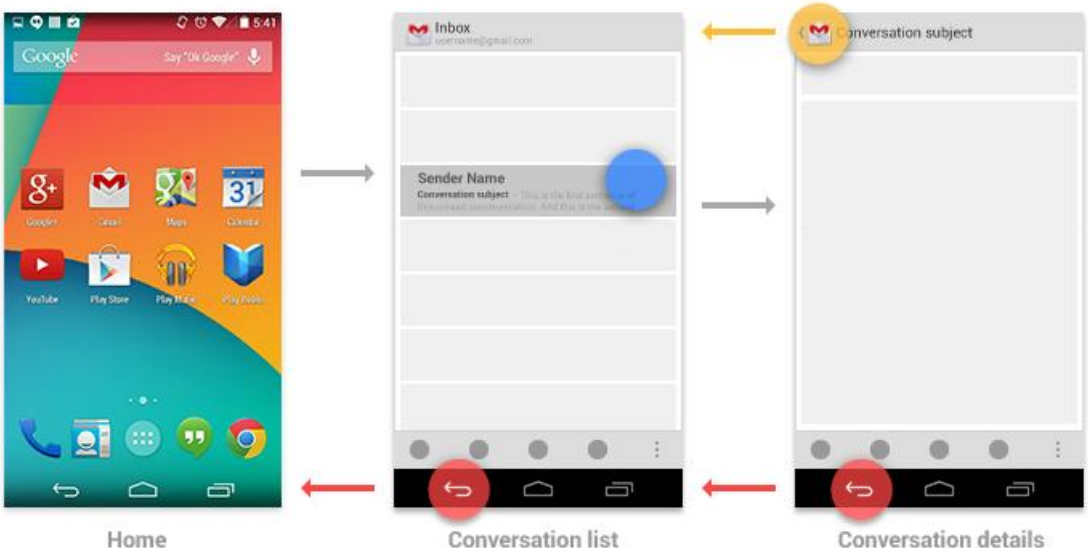
Внутренне непротиворечивая навигация является важнейшей составляющей пользовательского интерфейса. Мало что так раздражает пользователей, как несогласованное или непредсказуемое поведение элементов навигации.

Кнопка "Вверх" используется для навигации внутри приложения по иерархической структуре его экранов. Например, если на экране A отображается некоторый список и при выборе какого-либо элемента открывается экран B (с подробной информацией об этом элементе), то на экране B должна присутствовать кнопка "Вверх" для возврата к экрану A.



Если экран является самым верхним в приложении (то есть главным), он не должен содержать кнопку "Вверх".

Системная кнопка "Назад" используется для навигации в обратном хронологическом порядке среди экранов, недавно открытых пользователем. Такая навигация основана на порядке появления экранов, а не на иерархии приложения.

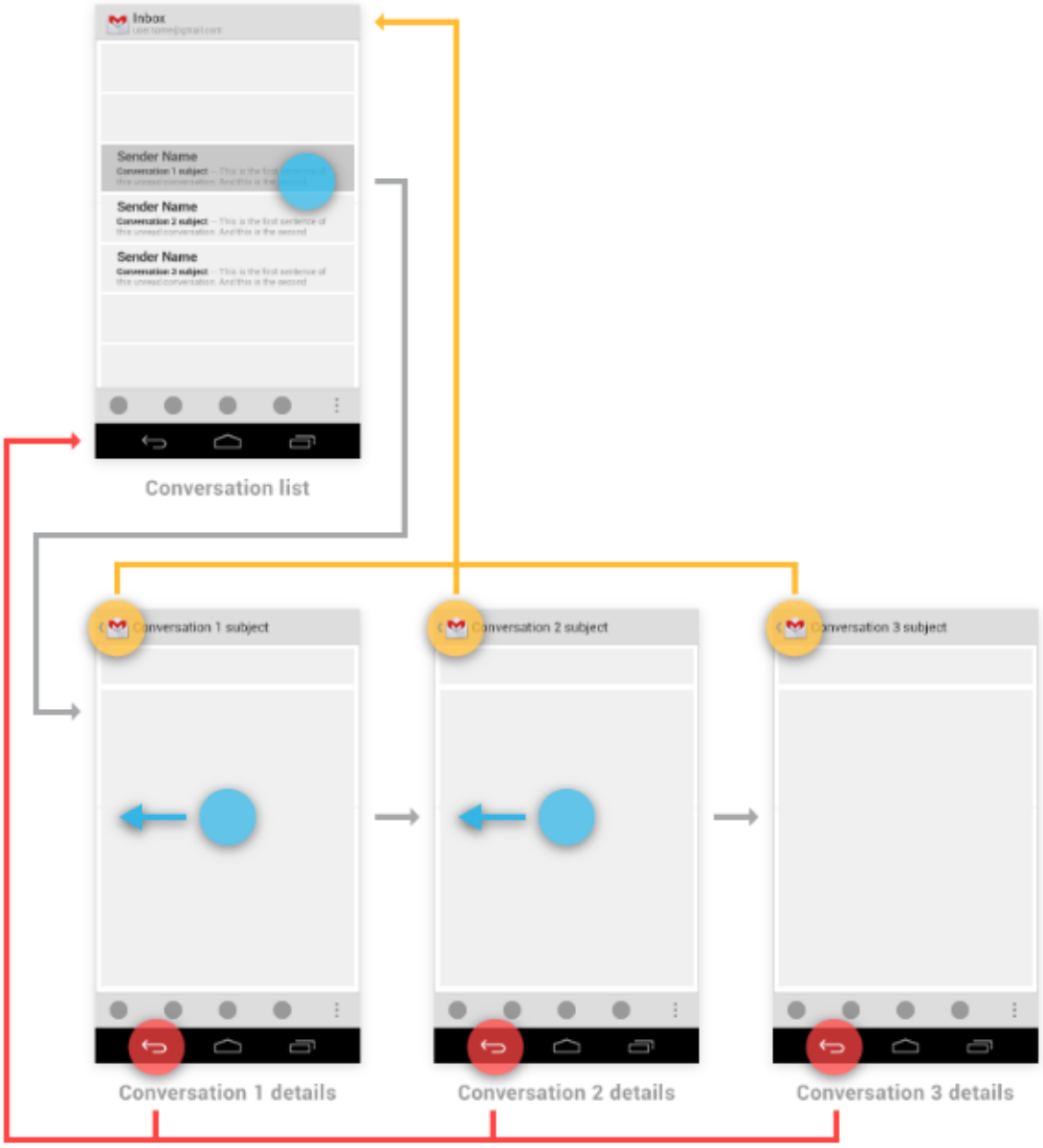


Если предыдущий экран одновременно является иерархическим родителем текущего, кнопка "Назад" имеет то же действие, что и кнопка "Вверх", — и это случается довольно часто.

Однако, в отличие от кнопки "Вверх", гарантирующей, что пользователь остается в приложении, кнопка "Назад" может перевести его на главный экран или даже в другое приложение.

Некоторые экраны не имеют строгой позиции в иерархии приложения, и на них можно перейти из нескольких точек. Например, на экран настроек можно попасть из любого другого экрана приложения. В таком случае кнопка "Вверх" должна осуществлять возврат на вызвавший экран, т. е. вести себя идентично кнопке "Назад".

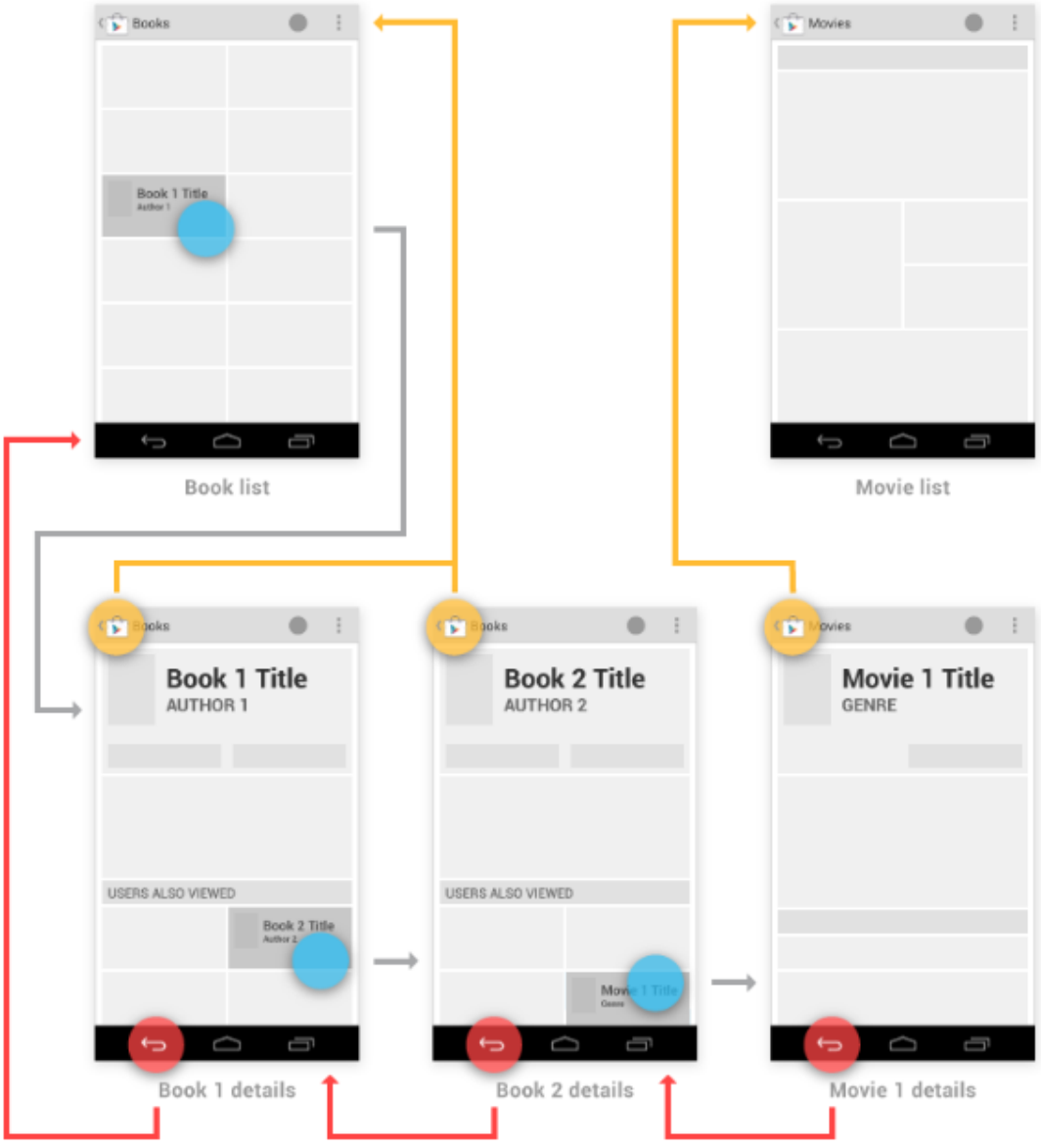
Когда приложение поддерживает навигацию от списка к подробному представлению одного из его элементов, нередко имеет смысл поддержать навигацию от этого элемента к другому, идущему в списке до или после него. Например, в Gmail можно провести пальцем влево или вправо по переписке, чтобы просмотреть предыдущие или последующие входящие сообщения. Как и в случае изменения представления на экране, такая навигация не меняет поведение кнопок "Вверх" и "Назад".



Однако существует важное исключение из этого правила во время просмотра подробных представлений элементов, не связанных вместе ссылающимся списком, например приложений одного разработчика или

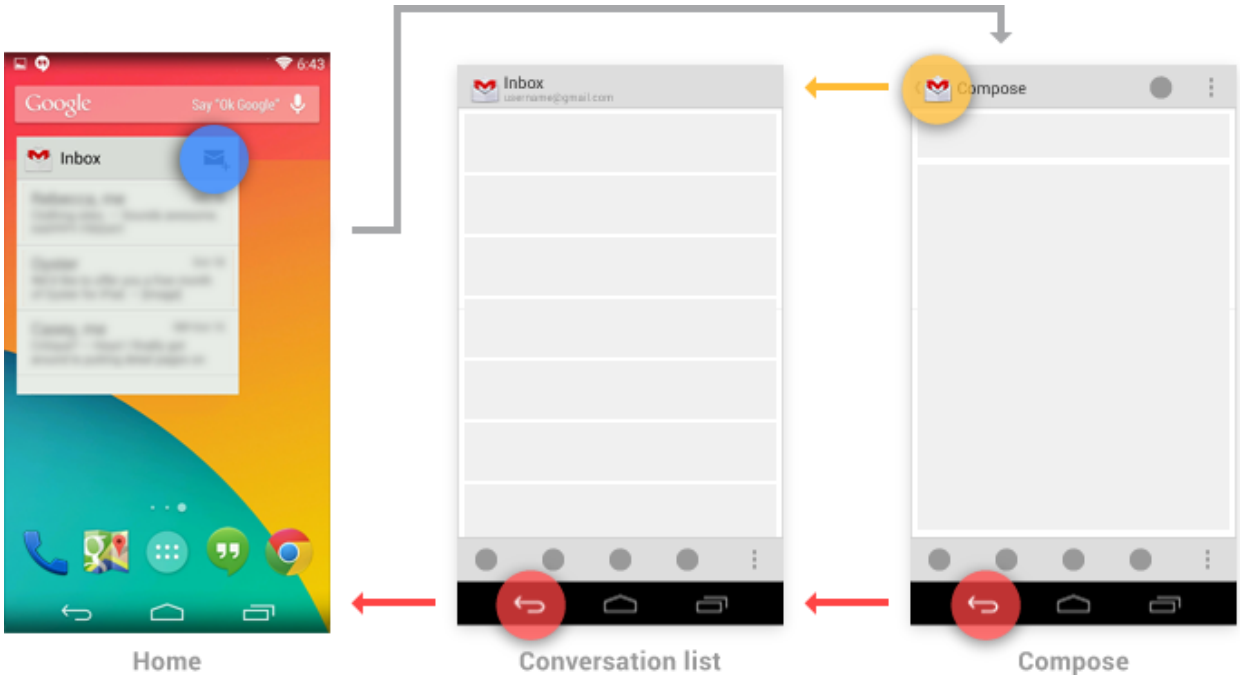
альбомов одного исполнителя в Play Store. В этих случаях переход по каждой ссылке создает историю навигации, что заставляет кнопку "Назад" перебирать все ранее просмотренные экраны. Кнопка "Вверх" должна по-прежнему игнорировать эти связанные по смыслу экраны и осуществлять переход на последний просмотренный контейнерный экран.

Разработчик может сделать поведение кнопки "Вверх" еще более интеллектуальным, исходя из своих знаний о структуре подробного представления. Вернемся к примеру с просмотром Play Store. Представьте, что пользователь перешел от информации о книге к информации о ее экранизации. Тогда кнопка "Вверх" могла бы переводить пользователя в контейнер (Фильмы), в котором он еще не был.



Навигация через виджеты приложений и уведомления

Вы можете использовать виджеты приложений или уведомления, чтобы помочь своим пользователям перейти непосредственно к экранам в пределах иерархии вашего приложения. Например, виджет «Входящие» Gmail и новое уведомление о сообщениях могут обойти экран «Входящие», непосредственно подключая пользователя к просмотру сеанса.



Если экран назначения достигается с одного конкретного экрана в вашем приложении, Up должен перейти к этому экрану.

В противном случае Up должен перейти к самому верхнему («Домашнему») экрану вашего приложения.

Одной из основных преимуществ системы Android является способность приложений активировать друг друга, предоставляя пользователю возможность перемещаться напрямую из одного приложения в другое. Например, приложение, которое должно захватить фотографию, может активировать приложение «Камера», которое вернет фотографию в соответствующее приложение.

Назначение родителя активности

Для назначения родительской активнности надо в файле манифеста AndroidManifest.xml

```
<activity
    android:name=".SendInfoActivity"
    android:label="@string/title_activity_send_info"
    android:parentActivityName=".MenuActivity"/>
```

Ваше приложение должно облегчить пользователям поиск пути к главному экрану приложения. Простой способ сделать это - предоставить кнопку «Вверх» на панели для всех активностей, кроме основной. Когда пользователь выбирает кнопку «Вверх», приложение переходит к родительской активности.

Чтобы включить кнопку «Вверх» для активности, которая имеет родительскую активность, вызовите метод **setDisplayHomeAsUpEnabled ()**. Как правило, вы делаете это, когда создается activity. Следующий метод onCreate () устанавливает панель инструментов и включает кнопку вверх на панели приложений:

```
protected void onCreate(Bundle savedInstanceState) {

    ActionBar actionBar = getActionBar();
    actionBar.setDisplayHomeAsUpEnabled(true);
}
```

Если если нет родителя - null.

Библиотеки поддержки Android

Support Library – библиотека, которая на старых версиях Android делает доступными возможности новых версий. Например, *фрагменты* появились только в третьей версии (API Level 11). Если вы хотите использовать их в своем приложении, это приложение не будет работать на более старых версиях Android, т.к. эти старые версии никогда не слышали про класс android.app.Fragment. Какие тут есть выходы?

1) Добавить в код проверку версии системы и в зависимости от результата выполнять тот или иной код. Т.е. если версия 11 и выше, используем фрагменты, иначе Activity. Вполне выполнимо, но не совсем просто. Можно ошибиться и запутаться. Т.е. при запуске приложения на старых версиях приходится либо отказываться от новшеств и пользоваться тем, что есть, либо изобретать велосипед и реализовывать новшества самому.

2) Можно позиционировать свое приложение только для новых версий. Тогда теряется ощутимая часть потенциальных пользователей вашей программы.

3) Использовать библиотеку *Support Library*. Она содержит классы - аналоги новшеств последних версий, которые будут работать на старых версиях.

уровень поддерживаемого API был изменен на Android 4.0 (уровень API 14) для всех пакетов поддержки библиотеки. Например, пакет поддержки-v4 и поддержки-v7 поддерживают минимальный уровень API, равный 14, для выпусков библиотеки поддержки от 26.0.0 и выше.

```
android.support.v*.
```

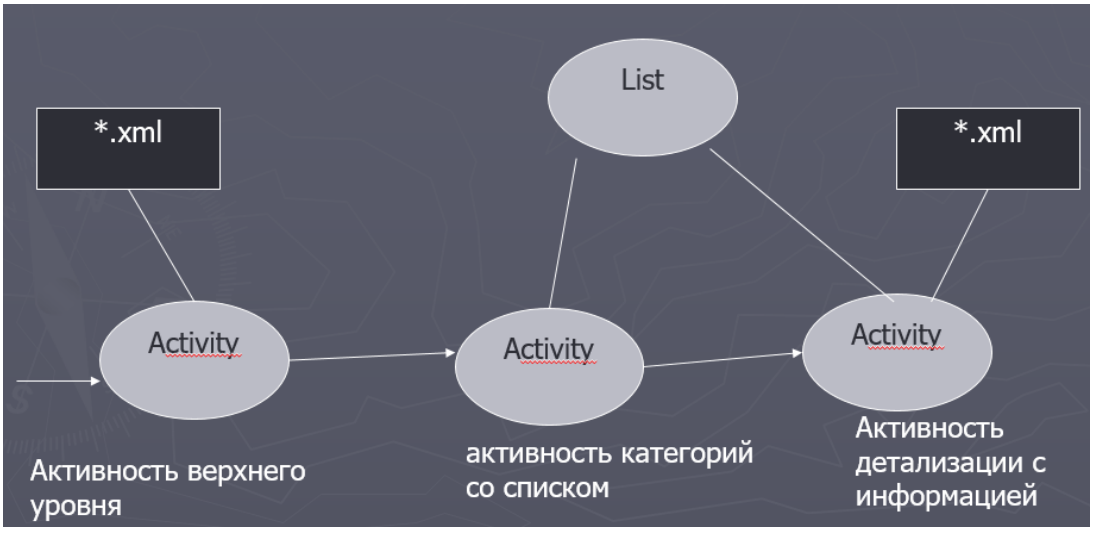
```
import android.support.v7.app.ActionBar;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;
```

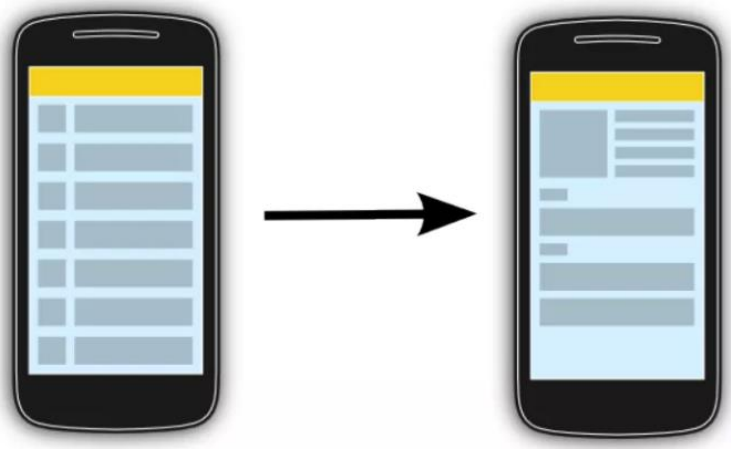
С выпуском Android 9.0 (API уровня 28) появилась новая версия библиотеки поддержки AndroidX, входящей в состав Jetpack. Библиотека AndroidX содержит существующую библиотеку поддержки, а также последние компоненты Jetpack.

Библиотеки эти периодически обновляются, в них добавляются новые классы, реализующие новые возможности. Так что, если вы не нашли в них сейчас то, что вам нужно, вполне возможно, что это появится в будущем.

Использование спискового представления для вывода списка

В соответсвии с описанием шаблонов навгации необходимо разработать следующий шаблон.





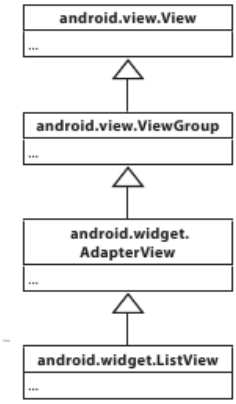
Пусть объектом для отображения в списке будет студент (сделаем списком в виде динамического массива)

```
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;
    }

    public String toString() {
        return this.name;
    }

    public String getName() {
        return name;
    }
```

Списковое представление позволяет вывести вертикальный список объектов данных, который в дальнейшем может использоваться для навигации по приложению. Добавим в макет списковое представление для набора команд, которые в дальнейшем будут открывать другие активности

Определение спискового представления в XML

Для добавления спискового представления в макет используется элемент <ListView>. Дерево наследования представлено ниже:



Чтобы заполнить списковое представление данными, используйте атрибут android:entries и присвойте ему массив строк. Строки из массива будут отображаться в списковом представлении в виде набора надписей TextView. Пример добавления в макет спискового представления, которое получает значения из массива строк:

```
<ListView
    android:id="@+id/list_ops"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:entries="@array/options"/>
```

Массив определяется точно так же, как это уже делалось ранее, — данные включаются в массив strings.xml:

```
<string-array name="options">
        <item>Students</item>
        <item>Listener</item>
    </string-array>

</resources>
```

Будет выведено:

Такой способ подходит только только для данных, представленных статическим массивом в strings.xml.

Чтобы пункты списка реагировали на щелчки, следует реализовать слушателя событий. Реализация слушателя событий позволит вам обнаруживать конкретные действия пользователя — скажем, щелчки на вариантах списка — и реагировать на них.

Если вы хотите, чтобы варианты списка реагировали на щелчки, создайте объект **OnItemClickListener** и реализуйте его метод **onItemClick()**. Слушатель **OnItemClickListener** отслеживает щелчки на вариантах списка, а метод **onItemClick()** определяет реакцию активности на щелчок. По параметрам, передаваемым методу **onItemClick(),** можно получить дополнительную информацию о событии — например, получить ссылку на вариант из списка, узнать его позицию в списковом представлении (начиная с 0) и идентификатор записи используемого набора данных. В нашем примере при щелчке на первом варианте спискового представления — варианте в позиции 0 — должна запускаться активность **StudentsCategoryActivity**. Если щелчок сделан на варианте в позиции 0, необходимо создать интент для запуска **ListenerCategoryActivity**. Код создания слушателя выглядит так:

```
AdapterView.OnItemClickListener itemClickListener =
                new AdapterView.OnItemClickListener()
                        {
 public void onItemClick(AdapterView<?> listView,
                                View itemView,
                                int position,
                                long id) {
                          if (position == 0) {
                           Intent intent = new Intent(TopList.this,
                                   StudentsCategoryActivity.class);
                            startActivity(intent);
                                    }
                           else {
                       Intent intent2 = new Intent(TopList.this,
                                    ListenerCategoryActivity.class);
                             startActivity(intent2);
                                    }
                                }
```

После того как слушатель будет создан, его необходимо добавить к ListView.

Назначение слушателя для спискового представления

После того как объект **OnClickItemListener** будет создан, его необходимо связать со списковым представлением. Эта задача решается при помощи метода **setOnItemClickListener()** класса **ListView**. Метод получает один аргумент — самого слушателя:

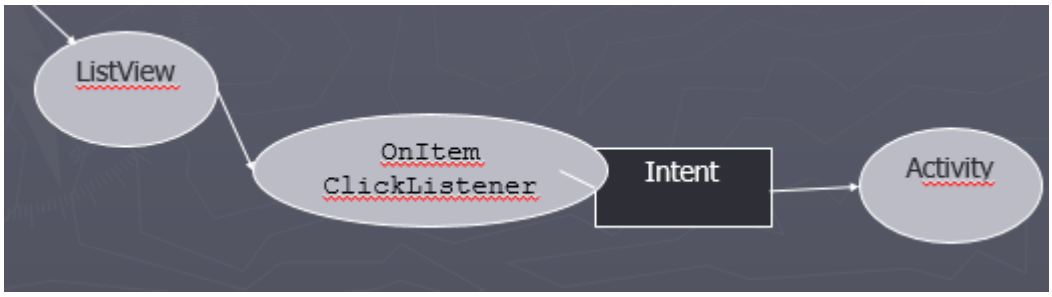
```
ListView listView = (ListView) findViewById(R.id.list_ops);
listView.setOnItemClickListener(itemClickListener);
```

Добавление слушателя к списковому представлению важно — именно эта операция обеспечивает получение слушателем оповещений о том, что пользователь щелкает на списковом представлении. Если этого не сделать, варианты из спискового представления не будут реагировать на щелчки.

Что происходит при выполнении кода

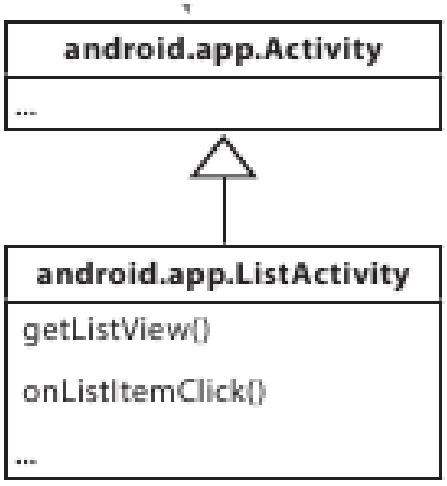
Метод **onCreate()** из **TopLevelActivity** создает объект **onItemClickListener** и связывает его с объектом **ListView**.

Когда пользователь щелкает на варианте из спискового представления, вызывается метод **onItemClick()** слушателя **onItemClickListener**. Если щелчок был сделан на Students, **onItemClickListener** создает интент для запуска активности **StudentsCategoryActivity**.



ListActivity

Списковая активность специализируется на работе со списком. Она автоматически связывается со списковым представлением, поэтому вам не придется создавать такое представление самостоятельно.



Списковые активности определяют свой макет на программном уровне, поэтому вам не придется создавать или заниматься сопровождением разметки

XML. Макет, генерируемый списковой активностью, содержит одно списковое представление. Для обращения к списковому представлению из кода активности используется метод **getListView()** списковой активности. Такое обращение необходимо для того, чтобы вы могли задать данные, которые должны выводиться в списковом представлении.

Вам не нужно реализовать собственного слушателя. Класс **ListActivity** уже реализует слушателя событий, который обнаруживает щелчки на вариантах спискового представления. Вместо того, чтобы создавать собственного слушателя и привязывать его к списковому представлению, разработчику достаточно реализовать метод **onListItemClick()** списковой активности. При таком подходе вам будет проще организовать реакцию активности на выбор вариантов спискового представления.

Ниже приведен базовый код создания списковой активности. Как видите, он очень похож на код создания обычной активности. Воспользуйтесь мастером New Activity для создания в проекте новой активности:

```
public class StudentsCategoryActivity extends ListActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```

Приведенный выше код создает базовую списковую активность. Так как класс представляет именно списковую активность, он должен расширять класс **ListActivity** вместо класса Activity.

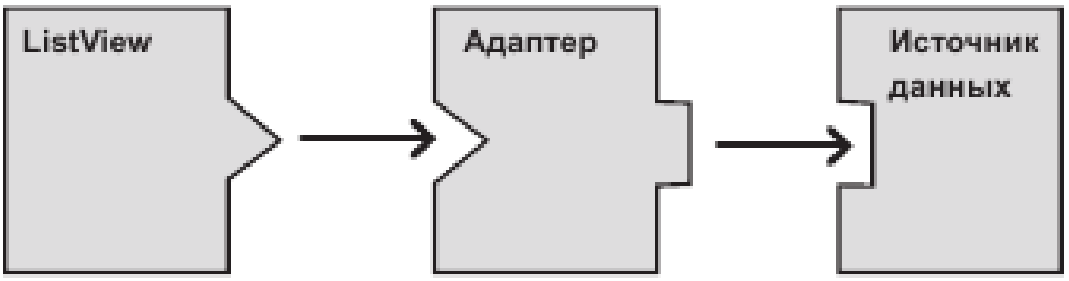
Другое отличие заключается в том, что вам не нужно назначать макет, используемый списковой активностью, вызовом **setContentView().** Дело в том, что списковые активности определяют свои макеты самостоятельно. Списковые активности, как и обычные, должны быть зарегистрированы в файле AndroidManifest.xml.

При создании первой активности **TopLevelActivity** мы могли связать данные со списковым представлением при помощи атрибута **android:entr**ies в XML макета. Такое решение работало, потому что данные хранились в виде ресурса статического массива строк. Массив был описан в файле strings.xml, что позволяло легко сослаться на него с использованием синтаксиса **android:entries="@array/options"**

А если в приложении используется другой способ хранения? Что, если данные хранятся в массиве, созданном на программном уровне в коде Java, или в базе данных? В этом случае атрибут android:entries не работает. Если списковое представление необходимо связать с данными, хранящимися в чем-то отличном от ресурса массива строк, придется написать код активности для

привязки данных. В нашем примере списковое представление требуется связать с массивом **Students** из класса **Student**.

Для нестатических данных используйте адаптер. Если данные спискового представления должны поступать из нестатического источника (например, из массива Java или базы данных), необходимо использовать адаптер. Адаптер играет роль моста между источником данных и списковым представлением:



Итак **Adapter** выступает в качестве посредника между источником данных и макетом **AdapterView** — **Adapter** извлекает данные (из источника, например, массива или запроса к базе данных) и преобразует каждую запись в представление, которое можно добавить в макет **AdapterView**.

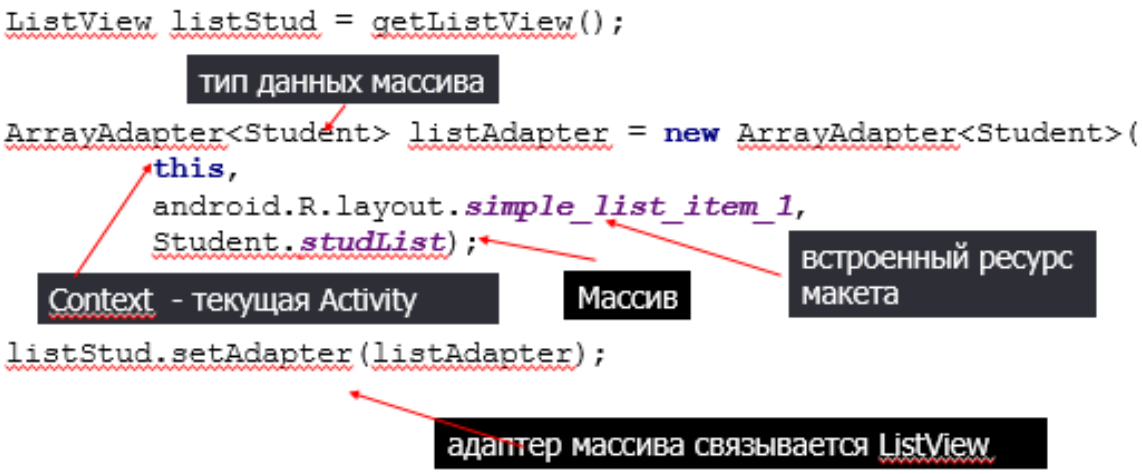
В Android предусмотрено несколько подклассов адаптера Adapter, которые полезно использовать для извлечения данных различных видов и создания представлений для **AdapterView**. Наиболее часто используемых адаптера: **ArrayAdapter и SimpleCursorAdapter**

ArrayAdapter

Этот адаптер используется в случае, когда в качестве источника данных выступает массив. По умолчанию **ArrayAdapter** создает представление для каждого элемента массива путем вызова метода toString() для каждого элемента и помещения его содержимого в объект TextView.

Адаптер массива может использоваться с любым субклассом класса AdapterView; это означает, что он будет работать как со списковым представлением, так и с раскрывающимся списком.

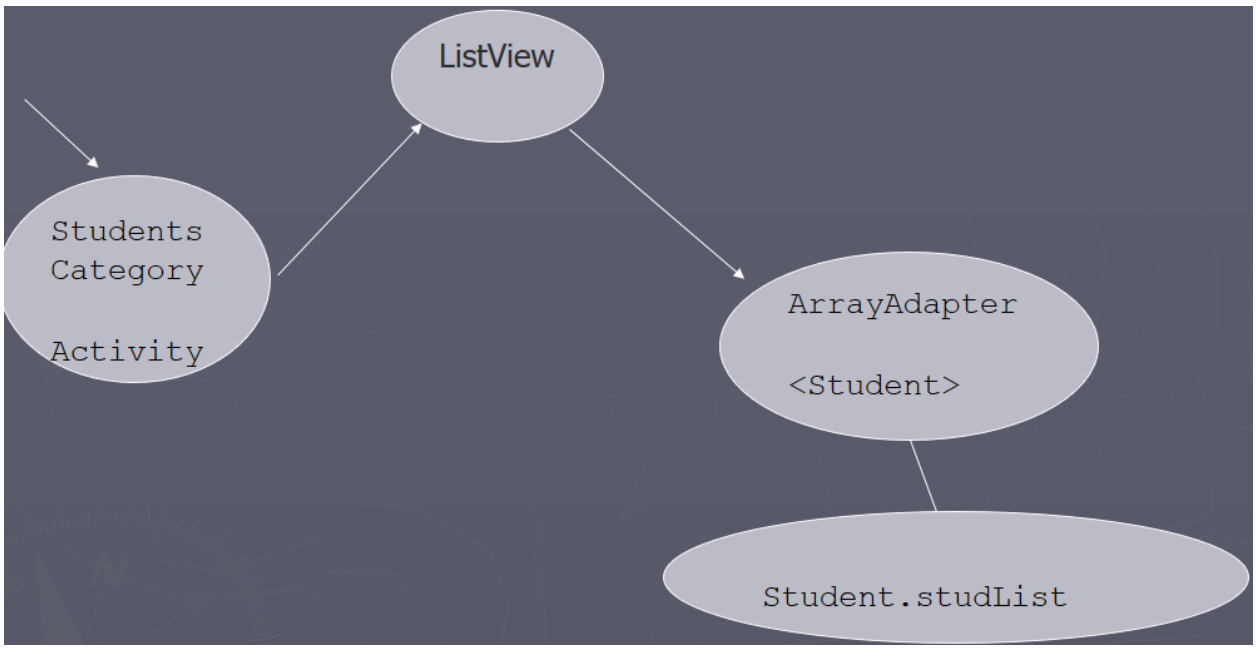
Чтобы использовать адаптер массива, следует инициализировать его и присоединить к списковому представлению. При инициализации адаптера массива вы сначала указываете тип данных массива, который вы хотите связать со списковым представлением. Затем адаптеру передаются три параметра: Context (обычно текущая активность), ресурс макета, который определяет, как должен отображаться каждый элемент из массива, и сам массив. Приведенный ниже код создает адаптер массива для отображения данных из массива студентов. Затем адаптер массива связывается со списковым представлением при помощи метода **setAdapter()** класса **ListView**:



Во внутренней реализации адаптер массива берет каждый элемент массива, преобразует его в String методом **toString()** и помещает каждый результат в надпись. Затем каждая надпись выводится в отдельной строке спискового представления.

Чтобы настроить внешний вид каждого элемента, можно переопределить метод toString() для объектов в массиве. Либо можно создать представление для каждого элемента, который отличается от **TextView** (например, если для каждого элемента массива требуется объект **ImageView**), наследовать класс **ArrayAdapter** и переопределить метод **getView(),** чтобы возвратить требуемый тип представления для каждого элемента.

Итак унас получилось:

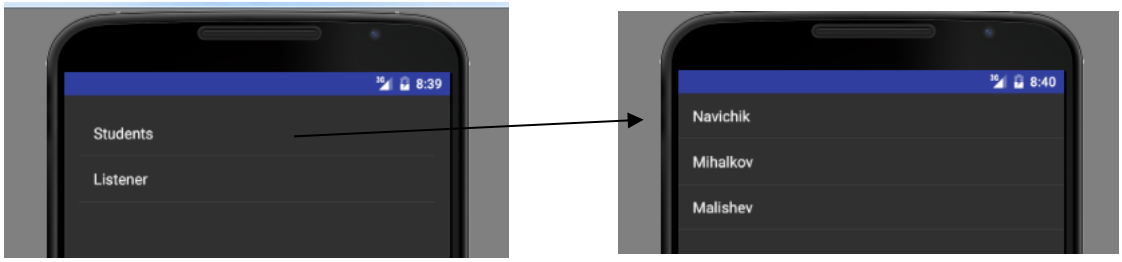


Когда пользователь выбирает команду, открывается активность StudentCategoryActivity. Так как StudentCategoryActivity является списковой активностью, она имеет макет по умолчанию с одним объектом **ListView**. Этот макет незаметно генерируется в коде Java, и он не определяется в разметке XML.

StudentCategoryActivity создает **ArrayAdapter<Student>** — адаптер массива для массивов, содержащих объекты **Student**.

Источником данных адаптера массива является массив students из класса **Student**. Для получения названия каждого объекта используется метод toString().

StudentCategoryActivity приказывает ListView использовать адаптер массива, вызывая метод **setAdapter()**. Списковое представление использует адаптер для вывода списка.



. Принципиальное отличие между Activity и ListActivity заключается в том, что класс **ListActivity** уже реализует слушателя событий щелчков. Вместо того, чтобы создавать собственного слушателя событий, при использовании списковой активности достаточно реализовать метод **onListItemClick()**

Метод **onListItemClick()** обычно используется для запуска другой активности, которая выводит подробное описание варианта, выбранного пользователем. Для этого разработчик создает интент, открывающий активность. Идентификатор варианта, выбранного пользователем, включается в дополнительную информацию, чтобы вторая активность могла использовать его при запуске. В нашем случае нужно запустить активность **StudentActivity** и передать ей идентификатор выбранного студента. **StudentActivity** использует эту информацию для вывода подробного описания напитка. Код выглядит так:

```
public class StudentsCategoryActivity extends ListActivity {

    public static final String EXTRA_NUM = "-1";
…
```

```
@Override
public void onListItemClick(ListView listView,
                            View itemView,
                            int position,
                            long id) {
    Intent intent = new Intent
          (StudentsCategoryActivity.this,
           StudentActivity.class);
    intent.putExtra(
               StudentyActivity.EXTRA_NUM, (int) id);
    startActivity(intent);
}
```

Передача идентификатора варианта, на котором был сделан щелчок, — практика весьма распространенная, так как передаваемое значение одновременно является идентификатором в используемом наборе данных. Если набор данных хранится в массиве, то идентификатор совпадает с индексом элемента массива. Если информация хранится в базе данных, то идентификатор является индексом записи в таблице. При подобном способе передачи идентификатора второй активности будет проще получить подробную информацию о данных, а затем вывести ее.

Создание Activity для детализации информаци

Как упоминалось ранее, **StudentActivity** является активностью детализации. Такие активности выводят подробную информацию о конкретной записи, и обычно переход к ним осуществляется из активностей категорий.

Добавьте в проект новую активность с именем StudentActivity и макет с именем activity_student. Ниже приведена разметка макета.

```
    <TextView
        android:id="@+id/name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/description"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
    <TextView
        android:id="@+id/rate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</RelativeLayout>
```

При запуске активность детализации читает из интента дополнительную информацию и использует ее для заполнения своих представлений.

Информация из интента используется для получения данных, которые должны выводиться в подробном описании.

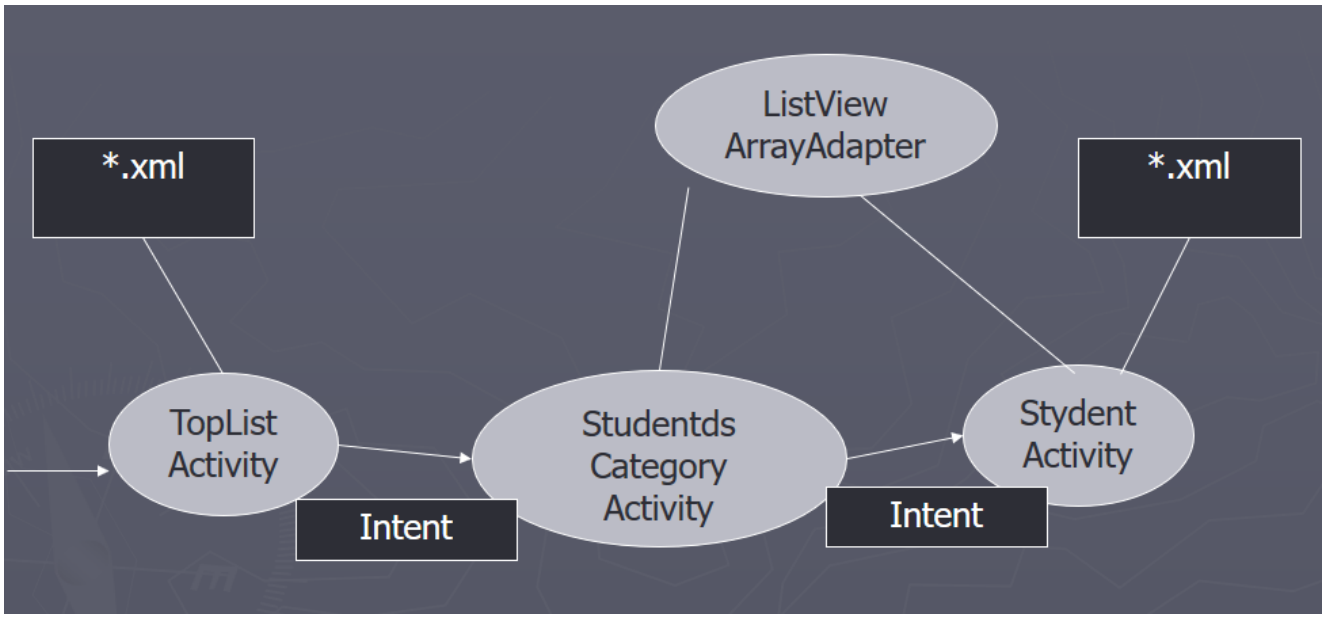
```
public static final String EXTRA_NUM = "-1";

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

     //Получить из данных интента
    int studNum =
(Integer)getIntent().getExtras().get(EXTRA_NUM);

     Student st= Student.studList[studNum];
```

При обновлении представлений в активности детализации необходимо позаботиться о том, чтобы отображаемые в них значения соответствовали данным, полученным из интента

```
TextView name = (TextView)findViewById(R.id.name);
name.setText(st.getName());

TextView description =
(TextView)findViewById(R.id.description);
description.setText(st.getInfo());

TextView rate = (TextView)findViewById(R.id.rate);
description.setText(st.getRate());
```

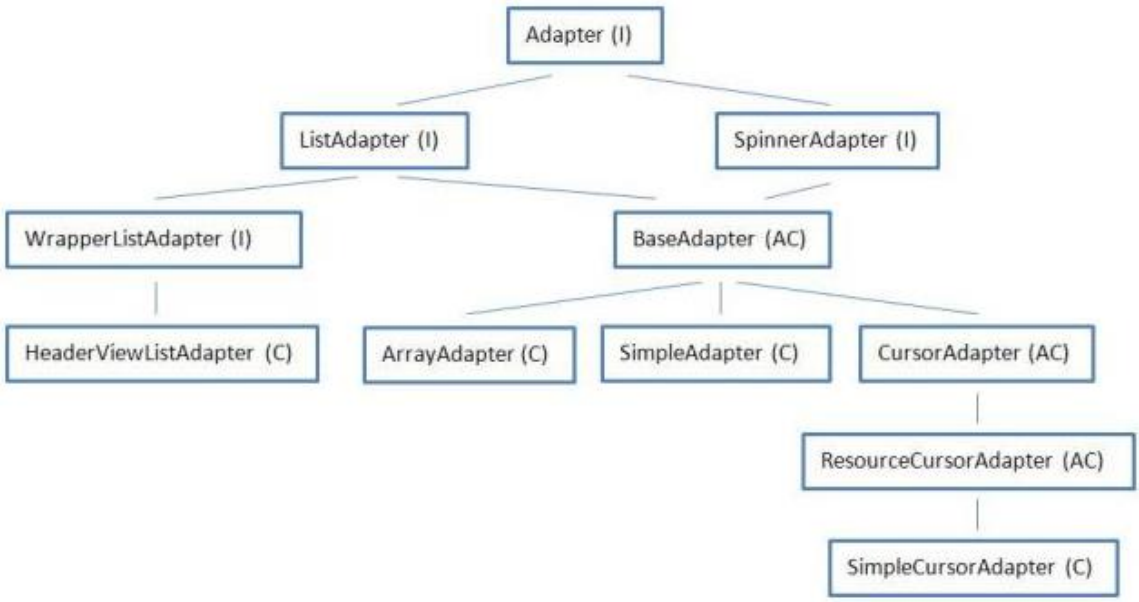
При запуске приложения открывается активность **TopListActivity**

Запускается активность **StudentActivity**. Активность читает номер из интента и получает подробную информацию о студенте из класса **Student**. Информация используется для обновления содержимого представлений.



Обзор адаптеров

Можно представить схему java-иерархии интерфейсов и классов адаптеров. В скобках тип: I – это интерфейс, AC – абстрактный класс, C – класс. Линии –наследование. Читать следует сверху вниз.



Интерфейс Adapter. Описывает базовые методы, которые должны содержать адаптеры: getCount, getItem, getView и пр.

Интерфейс ListAdapter. Этот интерфейс должен быть реализован адаптером, который будет использован в ListView (метод setAdapter). Содержит описание методов для работы с разделителями(separator) списка.

Интерфейс SpinnerAdapter. Адаптеры, реализующие этот интерфейс, используются для построения Spinner(выпадающий список или drop-down). Содержит метод getDropDownView, который возвращает элемент выпадающего списка.

Интерфейс WrapperListAdapter. Адаптеры, наследующие этот интерфейс используются для работы с вложенными адаптерами. Содержит метод getWrappedAdapter, который позволяет вытащить из основного адаптера вложенный.

Класс HeaderViewListAdapter. Готовый адаптер для работы с Header и Footer. Внутри себя содержит еще один адаптер (ListAdapter), который можно достать с помощью выше рассмотренного метода getWrappedAdapter из интерфейса WrapperListAdapter.

Абстрактный класс BaseAdapter. Содержит немного своих методов и реализует методы интерфейсов, которые наследует, но не все. Своим наследникам оставляет на обязательную реализацию методы: getView, getItemId, getItem, getCount из ListAdapter. Т.е. если хотите создать свой адаптер – это класс вам подходит.

Класс ArrayAdapter<T>. Готовый адаптер, который мы уже использовали. Принимает на вход список или массив объектов, перебирает его и вставляет строковое значение в указанный TextView. Кроме наследуемых методов содержит методы по работе с коллекцией данных – add, insert, remove, sort, clear и метод setDropDownViewResource для задания layout-ресурса для отображения пунктов выпадающего списка.

Класс SimpleAdapter. Также готовый к использованию адаптер. Принимает на вход список Map-объектов, где каждый Map-объект – это список атрибутов. Кроме этого на вход принимает два массива – from[] и to[]. В to указываем id экранных элементов, а в from имена(key) из объектов Map, значения которых будут вставлены в соответствующие (из from) экранные элементы.

Т.е. SimpleAdapter – это расширенный ArrayAdapter. Если вы делаете ListView и у вас каждый пункт списка содержит не один TextView, а несколько, то вы используете SimpleAdapter. Кроме наследуемых методов SimpleAdapter содержит методы по наполнению View-элементов значениями из Map – setViewImage, setViewText, setViewBinder. Т.е. видим, что он умеет работать не только с текстом, но и с изображениями. Метод setViewBinder – отличная штука, позволяет вам написать свой парсер значений из Map в View-элементы и адаптер будет использовать его.

Также содержит реализацию метода setDropDownViewResource.

Абстрактный класс CursorAdapter. Реализует абстрактные методы класса BaseAdapter, содержит свои методы по работе с курсором и оставляет наследникам методы по созданию и наполнению View: newView, bindView.

Абстрактный класс ResourceCursorAdapter. Содержит методы по настройке используемых адаптером layout-ресурсов. Реализует метод newView из CursorAdapter.

Класс SimpleCursorAdapter. Готовый адаптер, похож, на SimpleAdapter. Только использует не набор объектов Map, а Cursor, т.е. набор строк с полями. Соответственно в массив from[] вы заносите наименования полей, значения которых хотите вытащить в соответствующие View из массива to.
Содержит метод convertToString, который возвращает строковое значение столбца, который задается методом setStringConversionColumn. Либо можно задать свой конвертер методом setCursorToStringConverter и адаптер будет использовать его при вызове convertToString. В этом конвертере вы уже сами реализуете, что он будет возвращать.

Итого мы получили 4 готовых адаптера: HeaderViewListAdapter, ArrayAdapter<T>, SimpleAdapter, SimpleCursorAdapter.. Если у вас есть массив строк - то, не раздумывая, берете ArrayAdapter. Если работаете с БД и есть курсор, данные из которого надо вывести в список - используете SimpleCursorAdapter.

Если же эти адаптеры вам не подходят, есть набор абстрактных классов и интерфейсов, которые можем наследовать и реализовать в своих классах для создания своего адаптера. Да и готовые адаптеры всегда можно наследовать и сделать свою реализацию методов.

Кроме этой иерархии есть почти аналогичная ей. В ней содержатся адаптеры для работы с деревом – **ExpandableListView**. В целом они похожи на, уже рассмотренные. Но есть разница в том, что здесь данные не одноуровневые, а делятся на группы и элементы.