

ПРОГРАММИРОВАНИЕ МОБИЛЬНЫХ  
ПРИЛОЖЕНИЙ ПОД ПЛАТФОРМУ

# ANDROID



# Урок №4

Детальный  
обзор виджетов  
Android

## Содержание

### 1. Понятие Фрагмента.

Класс android.app.Fragment ..... 4

    1.1. Жизненный цикл Фрагмента ..... 5

    1.2. Добавление Фрагмента в Активность  
        с помощью объявления в файле  
        макета Активности ..... 9

    1.3. Добавление Фрагмента в Активность  
        программно ..... 16

    1.4. Зависимость жизненного цикла  
        Фрагмента от жизненного цикла Активности ..... 32

    1.5. Методы класса  
        android.app.FragmentManager ..... 44

1.6. "Стек Переходов Назад" (Back Stack) для Фрагментов . . . . .	45
1.7. Запуск Фрагмента без пользовательского интерфейса . . . . .	55
1.8. Фрагмент "Диалоговое окно". Класс android.app.DialogFragment . . . . .	63
1.9. Фрагмент "Список". Класс android.app.ListFragment . . . . .	90
<b>2. Понятие Intent. Класс android.content.Intent. Запуск Активностей других приложений . . . . .</b>	<b>102</b>
<b>3. Создание второй Активности в приложении и ее запуск . . . . .</b>	<b>125</b>
<b>4. Взаимодействие Android приложений с СУБД SQLite . . . . .</b>	<b>153</b>
4.1. Класс android.database.sqlite. SQLiteOpenHelper . . . . .	155
4.2. Класс android.content.ContentValues. . . . .	163
4.3. Класс android.database.sqlite.SQLiteDatabase . .	165
4.4. Класс android.database.sqlite.SQLiteCursor . .	174
4.5. Пример Android приложения, взаимодействующего с Базой Данных . . . . .	178
4.6. Адаптер Данных, класс android.widget. SimpleCursorAdapter . . . . .	206
4.7. Пример взаимодействия Android приложения с многотабличной БД и управления изменением версий Базы Данных . .	225
<b>5. Домашнее задание . . . . .</b>	<b>245</b>

# 1. Понятие Фрагмента. Класс android.app.Fragment

Фрагмент представляет из себя объект, который встраивается в Активность в виде отдельного модуля и является частью пользовательского интерфейса. У фрагмента есть свой жизненный цикл, который напрямую зависит от жизненного цикла Активности. В Активность можно встраивать несколько Фрагментов, причем в процессе работы приложения можно удалять ненужные Фрагменты и создавать новые и встраивать их в Активность вместо удаленных Фрагментов. Главное преимущество в применении Фрагментов заключается в том, что нет необходимости нагружать Активность большим набором разной функциональности — можно распределить функциональность приложения между разными Фрагментами, тем самым получив ту гибкость, которой обладают модульные приложения. Использование Фрагментов при разработке приложений так же позволяет легко применять одни и те же Фрагменты на разных этапах работы одного приложения, а так же переносить ранее созданный программный код Фрагментов в другие разрабатываемые приложения. Это возможно из-за того, что каждый фрагмент имеет свой собственный xml макет и собственное поведение со своими обратными вызовами жизненного цикла, которое (поведение) описывается в отдельном классе — классе Фрагмента.

Как уже упоминалось, Фрагмент встраивается в Активность, и жизненный цикл Активности влияет на жизненный

цикл каждого Фрагмента. Например, если Активность приостановлена, то в приостановленном состоянии находятся и все Фрагменты. Если Активность уничтожается, то уничтожаются и все принадлежащие ей Фрагменты.

Фрагмент представлен классом [android.app.Fragment](#). Иерархия для класса [android.app.Fragment](#) имеет следующий вид:

```
java.lang.Object  
|  
+-- android.app.Fragment
```

Фрагменты встраиваются в виджеты контейнеры (менеджеры раскладки, производные от класса [android.view.ViewGroup](#)). Встроить Фрагмент в Активность можно двумя способами:

- В xml макете Активности с помощью элемента [`<fragment>`](#).
- Программным способом.

Еще добавим, что Фрагмент не обязательно должен быть частью макета Активности. Фрагмент может вообще не иметь пользовательского интерфейса — в таком случае Фрагмент запускается как бы в виде Службы.

В данном уроке мы рассмотрим все способы создания Фрагментов.

## 1.1. Жизненный цикл Фрагмента

Перед тем как рассмотреть способы создания Фрагментов и встраивания их в Активность, рассмотрим жизненный цикл Фрагмента, т. е. последовательность событий, через которую проходит Фрагмент в процессе

работы приложения. Жизненный цикл Фрагмента изображен на Рис. 1.1.

Как видно из Рис. 1.1, Фрагмент на протяжении своего жизненного цикла проходит через такие события:

- **onAttach** — вызывается один раз, когда Фрагмент прикрепляется к Активности. Для обработки события необходимо в классе, производном от `android.app.Fragment` переопределить метод `void onAttach(Context context)`.
- **onCreate** — вызывается для инициализации Фрагмента. Для обработки события необходимо в классе, производном от класса `android.app.Fragment` переопределить метод `void onCreate(Bundle savedInstanceState)`. В этом методе, необходимо инициализировать ключевые компоненты Фрагмента, которые требуется сохранить, когда фрагмент находится в состоянии паузы или возобновлен после остановки.
- **onCreateView** — вызывается чтобы создать и вернуть (передать) в Активность набор виджетов, которые должны находятся в Фрагменте. Для обработки события необходимо переопределить метод `View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)`.
- **onActivityCreated** — событие вызывается, чтобы уведомить Фрагмент о том, что Активность, к которой он прикреплен, завершила обработку своего события `Activity.onCreate()`. Для обработки события `onActivityCreated` необходимо в классе, производном от класса `android.app.Fragment`, переопределить метод `void onActivityCreated(Bundle savedInstanceState)`.

- **onViewStateRestored** — событие сообщает Фрагменту, что все ранее сохраненные состояния виджетов из набора виджетов Фрагмента были восстановлены. Для обработки события необходимо переопределить метод **void onViewCreated(Bundle savedInstanceState)**.
- **onStart** — фрагмент становится видимым для пользователя. Для обработки события необходимо переопределить метод **void onStart()**.
- **onResume** — фрагмент становится доступным для взаимодействия с пользователем. Для обработки события необходимо переопределить метод **void onResume()**.
- **onPause** — фрагмент больше не взаимодействует с пользователем, потому что Активность приостановлена или происходят операции изменения (удаление или замена) над Фрагментом в Активности. Для обработки события необходимо переопределить



**Рис. 1.1. Жизненный цикл Фрагмента**

метод **void onPause()**. Именно в этом методе необходимо сохранять все состояния, которые должны быть сохранены за рамками текущего сеанса работы пользователя, так как пользователь может не вернуться назад.

- **onStop** — фрагмент более не виден пользователю, потому что Активность приостановлена или Фрагмент удаляется или заменяется в Активности. Для обработки события необходимо переопределить метод **void onStop()**.
- **onDestroyView** — событие предназначено для того чтобы Фрагмент освободил все ресурсы связанные с его набором виджетов. Для обработки события необходимо переопределить метод **void onDestroyView()**.
- **onDestroy** — вызывается для выполнения действий по окончательному завершению работы Фрагмента. Для обработки события необходимо переопределить метод **void onDestroy()**.
- **onDetach** — вызывается непосредственно перед тем, как фрагмент будет откреплен от Активности. Для обработки события необходимо переопределить метод **void onDetach()**.

Все перечисленные методы жизненного цикла Фрагмента очень тесно связаны (и зависят) от событий жизненного цикла Активности. Пример, показывающий зависимость жизненного цикла Фрагмента от жизненного цикла Активности, рассматривается ниже в данном уроке.

## 1.2. Добавление Фрагмента в Активность с помощью объявления в файле макета Активности

Как ранее уже говорилось, Фрагмент добавляет в Активность часть пользовательского интерфейса (набор виджетов Фрагмента) и этот интерфейс встраивается в общую иерархию виджетов Активности. В этом разделе мы рассмотрим способ добавления Фрагмента через объявление в файле макета Активности. Это наиболее часто используемый способ создания и встраивания Фрагментов.

Для того чтобы создать Фрагмент с помощью объявления в файле макета Активности, необходимо выполнить три шага. Первым шагом является создание xml файла с макетом, содержащим виджеты, которые будут принадлежать пользовательскому интерфейсу создаваемого Фрагмента. Создадим такой файл и назовем его /res/layout/first\_fragment.xml. Код xml разметки этого макета приведен в Листинге 1.1.

**Листинг 1.1.** Код xml разметки макета для Фрагмента из рассматриваемого примера

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"

    android:layout_margin="4dp"
    android:background="#FFFF00">
```

```
<EditText  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id/edtFirst" />  
  
<Button  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="Click Me!"  
    android:textAllCaps="false"  
    android:id="@+id/btnFirst" />  
  
</LinearLayout>
```

Как видно из Листинга 1.1, макет с виджетами пользовательского интерфейса для Фрагмента состоит из двух виджетов: текстового поля ([android.widget.EditText](#) который имеет идентификатор `edtFirst`), в который пользователь будет вводить текст, и кнопки ([android.widget.Button](#) с идентификатором `btnFirst`), по нажатию на которую функциональностью Фрагмента будет отображаться в Тосте введенный в текстовое поле `edtFirst` текст. Внешний вид Фрагмента можно увидеть на Рис. 1.2 (Для наглядности, главный контейнер Фрагмента имеет желтый цвет фона).

Как уже упоминалось, Фрагмент является объектом, а значит должен существовать класс, на основе которого этот объект будет создаваться. Поэтому вторым шагом является создание класса, производного от класса [android.app.Fragment](#) и добавление этого класса в проект. С помощью Android Studio создадим новый класс **FirstFragment** (который будет находиться в файле

/java/com.itstep.myapp/FirstFragment.java). Код класса **FirstFragment** представлен в Листинге 1.2.

**Листинг 1.2.** Класс FirstFragment для создаваемого  
в этом разделе Фрагмента

```
package com.itstep.myapp;
import ...

/**
 * Class FirstFragment
 *-----
 */
public class FirstFragment    extends Fragment
{
    //--- Class constants -----
    private final static String TAG =
        "===== FirstFragment";

    //--- Class Methods -----
    @Override
    public View onCreateView(LayoutInflater inflater,
                            ViewGroup container,
                            Bundle savedInstanceState)
    {
        Log.d(TAG, "===== onCreateView");

        //--- Создание с помощью LayoutInflater макета
        //--- с виджетами для Фрагмента
        View v = inflater.inflate(R.layout.
            first_fragment, container, false);

        //--- Назначаем обработчик события нажатия на
        //--- кнопку "Click Me!"
        Button btnFirst = (Button)
            v.findViewById(R.id.btnFirst);
```

```

btnFirst.setOnClickListener(
        new View.OnClickListener()
    {
        @Override
        public void onClick(View v)
        {
            Log.d(TAG, "===== onClick");

        //--- Находим текстовое поле, чтобы прочитать
        //--- из него текст -----
            EditText edtFirst = (EditText)
                FirstFragment.this.
                    getActivity().
                        findViewById(R.id.edtFirst);

        //--- Читаем введенный текст и выводим его в Тосте
            Toast.makeText(FirstFragment.this.
                getActivity(),
                edtFirst.getText().toString(),
                Toast.LENGTH_LONG).show();
        }
    });
    return v;
}
}

```

- Как видно из Листинга 1.2, в классе Фрагмента (**FirstFragment**), имеющего пользовательский интерфейс, необходимо переопределить метод **onCreateView**, который должен вернуть контейнер с набором виджетов для Фрагмента:

```

View onCreateView (LayoutInflater inflater,
    ViewGroup container,
    Bundle savedInstanceState);

```

Метод принимает:

- Ссылку (параметр **inflater**) на объект [android.view.LayoutInflater](#), с помощью которого можно создать набор виджетов на основе xml файла макета (класс [android.view.LayoutInflater](#) рассматривался в одном из предыдущих уроков).
- Ссылку (параметр **container**) на родительский констейнер, к которому будет прикреплен контейнер с набором виджетов Фрагмента создаваемый этим методом.
- Ссылку (параметр **savedInstanceState**) на объект [android.os.Bundle](#), из которого, при необходимости, можно извлечь ранее сохраненное состояние Фрагмента (см. Жизненный цикл Фрагмента).

В методе **onCreateView** Листинга 1.2, создается набор виджетов для Фрагмента на основе файла макета /res/layout/first\_fragment.xml (см. Листинг 1.1). Также в методе **onCreateView** происходит назначение обработчика события нажатия на кнопку с идентификатором **btnFirst**. Ведь, исходя из концепции модульности Фрагментов, обрабатывать события от своих виджетов должен сам Фрагмент, а не Активность, к которой он принадлежит (хотя можно реализовать и такой вариант обработки событий — если есть такая необходимость).

И последним, третьим шагом, является объявление Фрагмента в xml файле макета Активности (см. Листинг 1.3).

### Листинг 1.3. Объявление Фрагмента в файле xml макета Активности

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.itstep.myapp.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:textSize="10pt"
        android:textColor="#003366"
        android:text="Модуль app. Фрагменты." />

    <Space
        android:layout_width="match_parent"
        android:layout_height="20dp" />

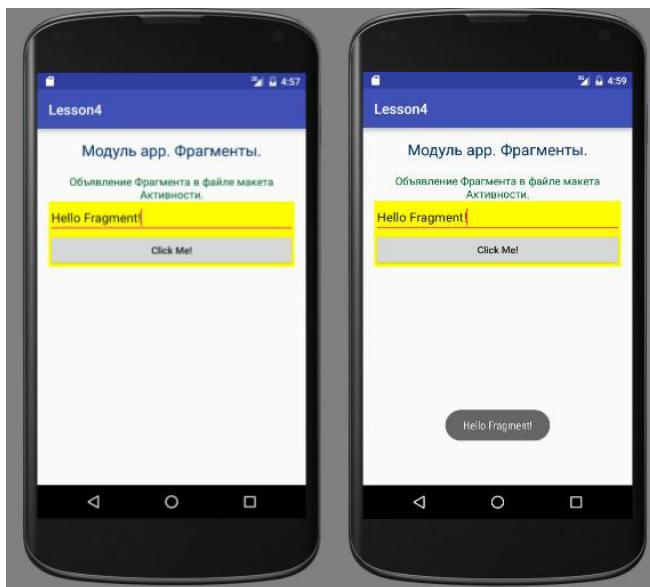
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:textSize="7pt"
        android:textColor="#006633"
        android:gravity="center_horizontal"
        android:text="Объявление Фрагмента в файле
        макета Активности." />

    <fragment
        android:name="com.itstep.myapp.FirstFragment"
```

```
    android:id="@+id/firstFragment"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    tools:layout="@layout/first_fragment" />

</LinearLayout>
```

В Листинге 1.3 жирным шрифтом выделен код объявления Фрагмента в xml разметке Макета Активности. Для этой цели используется элемент **<fragment>**, в котором кроме уже хорошо знакомых нам атрибутов **android:id**, **android:layout\_width**, **android:layout\_height** необходимо указать два обязательных атрибута **android:name** (значением для атрибута является имя класса, на основе которого будет создан объект Фрагмента) и **tools:layout**



**Рис. 1.2.** Внешний вид работы примера из Листингов 1.1, 1.2, 1.3

(значением атрибута является название файла с xml макетом пользовательского интерфейса Фрагмента).

Внешний вид приложения из Листингов 1.1, 1.2, 1.3 изображен на Рис. 1.2. Для наглядности, созданный Фрагмент имеет желтый цвет фона.

Фрагмент всегда может получить ссылку на Активность, к которой прикреплен с помощью метода (класс `android.app.Fragment`):

```
Activity getActivity();
```

Исходные коды примера, рассматриваемого в данном разделе можно найти в модуле «app» среди файлов с исходными кодами, которые прилагаются к данному уроку.

### 1.3. Добавление Фрагмента в Активность программно

Фрагмент можно добавить (встроить) в Активность и программным способом. Для этого необходимо создать Фрагмент программно и прикрепить его к специально существующему для этой цели виджету контейнеру (объекту класса, производного от [android.view.ViewGroup](#). Очень удобно для этой цели использовать виджет контейнер [android.widget.FrameLayout](#), который рассматривался в наших предыдущих уроках.

Последовательность шагов для данного способа добавления Фрагмента в Активность следующая: сначала создается xml файл с разметкой пользовательского интерфейса Фрагмента, затем создается java класс Фрагмента, далее в макете Активности создается контейнер, куда будет встроен Фрагмент, и последний шаг – это программный код встраивания Фрагмента в Активность, например

в обработчике события нажатия на кнопку или в методе `onCreate`. Давайте пройдем все эти шаги последовательно.

**ПЕРВЫЙ ШАГ.** Создадим xml файл с разметкой пользовательского интерфейса для нашего Фрагмента. В рассматриваемом примере это будет простейший калькулятор (функционал калькулятора нам понятен и позволит нам сконцентрироваться только на механизме добавления Фрагмента). Созданный файл с xml разметкой называется `/res/layout/calc_fragment.xml` и его содержимое представлено в Листинге 1.4.

**Листинг 1.4.** Xml верстка макета с виджетами пользовательского интерфейса для Фрагмента, рассматриваемого в данном разделе примера

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:background="#7ae2e2"
    android:padding="4dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:text="Калькулятор"
        android:textSize="9pt"
        android:textColor="#003366" />

    <LinearLayout
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:orientation="horizontal">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="7pt"
        android:textColor="#003366"
        android:text="Первое число: " />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/edtOne"
        android:inputType="numberDecimal" />

</LinearLayout>

<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="7pt"
        android:textColor="#003366"
        android:text="Второе число: " />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/edtTwo"
        android:inputType="numberDecimal" />

</LinearLayout>
```

```
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAllCaps="false"
        android:text="+"
        android:id="@+id	btnPlus" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAllCaps="false"
        android:text="-"
        android:id="@+id	btnMinus" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAllCaps="false"
        android:text="/"
        android:id="@+id	btnDiv" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAllCaps="false"
        android:text="*"
        android:id="@+id	btnMult" />

</LinearLayout>

<LinearLayout
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal">

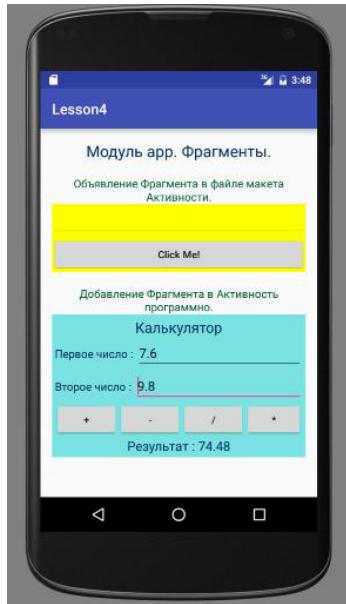
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="8pt"
        android:textColor="#003366"
        android:text="Результат: " />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="8pt"
        android:textColor="#003366"
        android:id="@+id/tvResult" />

</LinearLayout>
</LinearLayout>
```

Внешний вид Фрагмента (и всего примера, рассматриваемого в данном разделе) изображен на Рис. 1.3.

Как видно из Листинга 1.4, пользователь должен ввести первое число в текстовое поле (`android.widget.EditText`) с идентификатором `edtOne`. Для ввода второго числа предназначено текстовое поле (`android.widget.EditText`) с идентификатором `edtTwo`. Далее пользователь нажимает на одну из кнопок «Сложить», «Вычесть», «Разделить», «Умножить» (идентификаторы `btnPlus`, `btnMinus`, `btnDiv`, `btnMult` соответственно). Результат арифметической операции будет отображен в текстовом поле (`android.widget.TextView`) с идентификатором `tvResult`. В случае неправильного ввода чисел появится Тост с сообщением об исключительной ситуации.



**Рис. 1.3.** Внешний вид приложения из рассматриваемого в данном разделе примера

**ВТОРОЙ ШАГ.** Создадим класс для нашего Фрагмента. Класс будет называться **CalcFragment**. Найдется этот класс в файле `/java/com.itstep.myapp/CalcFragment.java`. Исходный код этого класса представлен в Листинге 1.5.

**Листинг 1.5.** Программный код класса `CalcFragment` для создаваемого в данном разделе Фрагмента

```
package com.itstep.myapp;
import ...
/**
 * Class CalcFragment – Фрагмент, реализующий
 * функциональность простейшего калькулятора
 * -----
 */
```

```
public class CalcFragment extends Fragment
        implements View.OnClickListener
{
    //--- Class constants -----
    private final static String TAG = "==== CalcFragment";

    //--- Class members -----
    /**
     * Текстовое поле, в которое пользователь
     * введет первое число
     */
    private EditText edtOne;

    /**
     * Текстовое поле, в которое пользователь
     * введет второе число
     */
    private EditText edtTwo;

    /**
     * Текстовое поле, в которое Фрагмент
     * выведет результат вычисления калькулятора
     */
    private TextView tvResult;

    //--- Class methods -----
    @Override
    public View onCreateView(LayoutInflater inflater,
                            ViewGroup container,
                            Bundle savedInstanceState)
    {
        Log.d(TAG, "----- onCreateView");

        //--- Создание набора виджетов пользовательского
        //--- интерфейса Фрагмента
        View view = inflater.inflate(R.layout.
                                    calc_fragment, container, false);
    }
}
```

```
//-- Инициализация полей объекта -----
    this.edtOne = (EditText)
        view.findViewById(R.id.edtOne);
    this.edtTwo = (EditText)
        view.findViewById(R.id.edtTwo);
    this.tvResult = (TextView)
        view.findViewById(R.id.tvResult);

//-- Назначение кнопкам Фрагмента обработчиков
//-- событий нажатия -----
    (view.findViewById(R.id.btnPlus)).
        setOnClickListener(this);
    (view.findViewById(R.id.btnMinus)).
        setOnClickListener(this);
    (view.findViewById(R.id.btnDiv)).
        setOnClickListener(this);
    (view.findViewById(R.id.btnMult)).
        setOnClickListener(this);

    return view;
}

/**
 * Обработчик событий нажатий на кнопки Калькулятора.
 */

@Override
public void onClick(View v)
{
    try
    {
//-- Считываем из Текстового поля edtOne первое
//-- введенное пользователем число -----
        double one = Double.parseDouble(
            this.edtOne.getText().toString());

//-- Считываем из Текстового поля edtTwo второе
//-- введенное пользователем число
```

```
        double two = Double.parseDouble(
            this.edtTwo.getText() .
            toString());

        //--- Делаем вычисление результата в зависимости
        //--- от нажатой кнопки калькулятора
        double result = 0;

        switch (v.getId())
        {
            case R.id.btnAdd:
                result = one + two;
                break;

            case R.id.btnMinus:
                result = one - two;
                break;

            case R.id.btnDiv:
                result = one / two;
                break;

            case R.id.btnMult:
                result = one * two;
                break;
        }

        //--- Выводим результат вычисления в текстовое
        //--- поле tvResult -----
        this.tvResult.setText(String.valueOf(result));
    }
    catch (Exception e)
    {
        Toast.makeText(this.getActivity(),
            e.getMessage(), Toast.LENGTH_SHORT).show();
    }
}
```

Как видно из Листинга 1.5, подход для реализации функциональности Фрагмента практически не отличается от подхода, который применяется для Активности. Основные действия по инициализации Фрагмента (как и Активности) осуществляются в методе **onCreateView** (для Активности это метод **onCreate**). Сам пример из Листинга 1.5 хорошо документирован комментариями, поэтому дополнительных пояснений к примеру приводить не будем.

**ТРЕТИЙ ШАГ.** Добавление в макет Активности контейнера, в который будет встроен наш Фрагмент с функциональностью калькулятора. На этом шаге тоже все достаточно просто и здесь для нас нет ничего нового. Код разметки макета Активности (файл /res/layout/activity\_main.xml) с контейнером приведен в Листинге 1.6.

**Листинг 1.6.** Контейнер android.widget.FrameLayout, в который будет программно встроен Фрагмент

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.itstep.myapp.MainActivity">

    ...
<Space
    android:layout_width="match_parent"
    android:layout_height="20dp"/>
```

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_horizontal"  
    android:textSize="7pt"  
    android:textColor="#006633"  
    android:gravity="center_horizontal"  
    android:text="Добавление Фрагмента  
    в Активность программно." />  
  
<FrameLayout  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id/flFragmentContainer">  
  
</FrameLayout>  
</LinearLayout>
```

В Листинге 1.6 выделен жирным шрифтом код контейнера **android.widget.FrameLayout**. Напомним, что в этот контейнер можно вставить только один дочерний виджет. Таковым дочерним виджетом и будет главный контейнер нашего Фрагмента. Напомним, что внешний вид встроенного Фрагмента изображен на Рис. 1.3.

**ЧЕТВЕРТЫЙ ШАГ**, последний. Программное встраивание Фрагмента в Активность. Осуществляется это с помощью объектов [android.app.FragmentManager](#) и [android.app.FragmentTransaction](#). Непосредственно добавление, удаление и замену Фрагментов в Активности выполняет объект класса [android.app.FragmentTransaction](#), но чтобы получить ссылку на этот объект, необходимо сначала получить ссылку на объект [android.app.FragmentManager](#).

Это можно сделать с помощью вызова метода класса Активности [android.app.Activity](#):

```
FragmentManager getFragmentManager ();
```

Этот метод возвращает ссылку на объект **android.app.FragmentManager**, с помощью которого можно взаимодействовать с Фрагментами данной Активности. Более подробно методы класса **android.app.FragmentManager** будут рассмотрены в данном уроке позже. В данный момент нас интересует только метод получения ссылки на объект **android.app.FragmentTransaction**:

```
FragmentTransaction beginTransaction();
```

В описании этого метода сообщается, что метод начинает последовательность операций по изменению Фрагментов в Активности для ассоциированного с ней объекта **android.app.FragmentManager**. Другими словами, объект **android.app.FragmentTransaction** является транзакцией, в течении которой можно изменять (добавлять, удалять, редактировать) Фрагменты в Активности. По завершении транзакции ее необходимо подтвердить с помощью вызова метода **commit()**. Давайте рассмотрим наиболее важные методы класса **android.app.FragmentTransaction**:

- **FragmentTransaction add(int containerViewId, Fragment fragment, String tag)** — добавляет Фрагмент **fragment** в виджет контейнер с идентификатором **containerViewId**, который расположен на Активности. Параметр **tag** позволяет задать строковое значение для добавляемого Фрагмента, которое может быть использовано,

например для поиска Фрагмента. Этот параметр **tag** является своеобразным идентификатором Фрагмента в Активности. Метод возвращает ссылку на текущий объект **android.app.FragmentTransaction**.

- **FragmentTransaction attach(Fragment fragment)** — прикрепляет ранее открепленный (с помощью метода **detach**) от пользовательского интерфейса Фрагмент **fragment**. Набор виджетов этого Фрагмента будет пересоздан, прикреплен к пользовательскому интерфейсу и отображен (см. Жизненный цикл Фрагмента). Метод возвращает ссылку на текущий объект **android.app.FragmentTransaction**.
- **int commit()** — подтверждает завершение транзакции. Изменения, произведенные в данной транзакции, вступают в силу не сразу — транзакция помещается на исполнение в качестве задачи главного потока приложения, которая будет исполнена, как только главный поток приложения освободится от выполнения других задач. Метод возвращает идентификатор транзакции в «Стеке Переходов Назад» (если данная транзакция добавлялась в «Стек Переходов Назад» с помощью вызова метода **addToBackStack**) или отрицательное значение, если транзакция в «Стек Переходов Назад» не добавлялась. Что такое «Стек Переходов Назад» будет рассказано в данном уроке позже. Однако, чтобы не держать интриги, забежим немного вперед и приоткроем тайну понятия «Стек Переходов Назад» — владельцы мобильных устройств с Операционной Системой Android знают, что если в исполняющемся приложении запускается еще одна

Активность, то текущая Активность не уничтожается, а остается доступной с помощью нажатия на кнопку «Назад» устройства. То есть, при запуске в приложении другой Активности, предыдущая Активность помещается в «Стек Переходов Назад», чтобы потом пользователь смог к ней вернуться через кнопку «Назад». Аналогично и для Фрагментов. Фрагмент можно поместить в «Стек Переходов Назад» чтобы иметь возможность вернуть Фрагмент обратно на Активность в том же состоянии, в котором Фрагмент был помещен в «Стек Переходов Назад». Так вот, идентификатор транзакции, который возвращает данный метод, может быть использован для извлечения изменений этой транзакции из «Стека Переходов Назад» с помощью метода **void popBackStack(int id, int flags)** класса **android.app.FragmentManager** (рассматривается ниже в данном уроке).

- **FragmentTransaction detach(Fragment fragment)** — открепляет Фрагмент **fragment** от пользовательского интерфейса. Это тоже самое, когда Фрагмент помещается в «Стек Переходов Назад» — фрагмент удаляется из пользовательского интерфейса, но остается доступным для взаимодействия посредством объекта **android.app.FragmentManager**. При этом набор виджетов Фрагмента уничтожается. Метод возвращает ссылку на текущий объект **android.app.FragmentTransaction**.
- **FragmentTransaction hide(Fragment fragment)** — прячет (скрывает, делает невидимым) фрагмент **fragment**. Метод возвращает ссылку на текущий объект **android.app.FragmentTransaction**.

- **FragmentTransaction remove(Fragment fragment)** — удаляет Фрагмент **fragment** из Активности. Метод возвращает ссылку на текущий объект **android.app.FragmentTransaction**.
- **FragmentTransaction replace(int containerViewId, Fragment fragment, String tag)** — заменяет текущий Фрагмент в контейнере **containerViewId** на другой Фрагмент **fragment** и задает добавляемому Фрагменту строковый идентификатор **tag**. Вызов этого метода аналогичен вызову метода **remove** и последующему вызову метода **add**. Метод возвращает ссылку на текущий объект **android.app.FragmentTransaction**.
- **FragmentTransaction setCustomAnimations(int enter, int exit)** — задает специальные эффекты анимации, которые будут применяться при смене (добавлении, удалении) Фрагментов. Параметры **enter** и **exit** — идентификаторы Анимаций из ресурсов приложения. Работа с Анимацией будет рассмотрена в следующих уроках.
- **FragmentTransaction setTransition(int transit)** — задает стандартную анимацию, которая будет применяться при смене Фрагментов. Параметр **transit** может принимать одно из следующих значений: **TRANSIT\_NONE** (анимации нет), **TRANSIT\_FRAGMENT\_OPEN** (анимация на добавление Фрагмента), **TRANSIT\_FRAGMENT\_CLOSE** (анимация на удаление Фрагмента), **TRANSIT\_FRAGMENT\_FADE** (анимация на добавление и удаление Фрагмента).
- **FragmentTransaction show(Fragment fragment)** — показывает ранее спрятанный с помощью вызова метода

**hide** Фрагмент **fragment**. Метод возвращает ссылку на текущий объект **android.app.FragmentManager**.

- **FragmentManager** **addBackStack (String name)** — добавляет текущую транзакцию в «Стек Переходов Назад». Параметр **name** позволяет задать имя для этой транзакции в стеке, может быть null. Имя может быть использовано для извлечения ее из «Стека Переходов Назад» с помощью метода **voidpopBackStack(String name, int flags)** класса **android.app.FragmentManager** (рассматривается ниже в данном уроке).

Исходя из описанных методов класса **android.app.FragmentManager**, последовательность действий для программного добавления Фрагмента в Активность является следующей: получение ссылки на **android.app.FragmentManager**, затем вызов метода **add** и подтверждение завершения транзакции с помощью вызова метода **commit**. Указанная последовательность действий для рассматриваемого примера приведена в Листинге 1.7 и осуществляется в методе Активности **onCreate**.

#### Листинг 1.7. Программное добавление Фрагмента в Активность

```

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

    //-- Добавление Фрагмента в Активность программно
    FragmentManager fragmentManager =
        this.getFragmentManager();
}

```

```
FragmentTransaction fragmentTransaction =  
    fragmentManager.beginTransaction();  
CalcFragment fragment = new CalcFragment();  
fragmentTransaction.add(R.id.flFragmentContainer,  
    fragment);  
fragmentTransaction.commit();  
}
```

В Листинге 1.7 создается объект Фрагмента **Calc Fragment** с помощью оператора new и затем прикрепляется к контейнеру **android.widget.FrameLayout** с идентификатором **flFragmentContainer** (см. Листинг 1.6). Внешний вид работы примера изображен на Рис. 1.3. Исходный код примера из данного раздела находится в модуле «app» файлов исходного кода, прилагаемых к данному уроку.

## 1.4. Зависимость жизненного цикла Фрагмента от жизненного цикла Активности

Теперь, когда мы научились создавать Фрагменты и встраивать их в Активность, вернемся еще раз к жизненному циклу Фрагментов, чтобы увидеть зависимость их жизненного цикла от жизненного цикла Активности, в которую они встроены.

Данный раздел важен для понимания функционирования Фрагментов.

Зависимость жизненного цикла рассмотрим на несложном примере. В этом примере мы переопределим все методы обратного вызова для жизненного цикла Активности (**onCreate**, **onPause** и т.д.) и все методы обратного вызова для встроенного в Активность Фрагмента. Все

переопределенные методы будут выводить в Лог приложения свое сообщение. Вот на основе этих лог сообщений мы и понаблюдаем за зависимостью жизненного цикла Фрагментов от жизненного цикла Активности.

Создадим файл (/res/layout/test\_fragment.xml) с xml версткой виджетов пользовательского интерфейса Фрагмента. Пользовательский интерфейс Фрагмента будет очень простой — одно текстовое поле **android.widget.TextView**. Содержимое файла /res/layout/test\_fragment.xml представлено в Листинге 1.8.

**Листинг 1.8.** Xml макет пользовательского интерфейса Фрагмента из примера текущего раздела

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:textColor="#f75e5e"
        android:textSize="7pt"
        android:text="Test Fragment" />

</LinearLayout>
```

Далее, создадим класс Фрагмента и назовем его **TestFragment** (файл /java/com.itstep.myapp/TestFragment.java). В этом классе переопределим ВСЕ методы обратного

вызыва для событий жизненного цикла Фрагмента. Код класса **TestFragment** приведен в Листинге 1.9.

**Листинг 1.9.** Код класса Фрагмента TestFragment рассматриваемого в данном разделе примера

```
package com.itstep.myapp;
import ...

/**
 * Class TestFragment – Наблюдение за жизненным
 * циклом Фрагмента по отношению к жизненному
 * циклу Активности, в которую он встроен.
 *
 */
public class TestFragment extends Fragment
{
    //--- Class constants -----
    private final static String TAG = "===== TestFragment";

    //--- Class methods -----
    @Override
    public View onCreateView(LayoutInflater inflater,
                            ViewGroup container,
                            Bundle savedInstanceState)
    {
        Log.d(TAG, "----- onCreateView –
               Создание набора виджетов Фрагмента");

        //--- Создание с помощью LayoutInflater макета
        //--- с виджетами для Фрагмента
        return inflater.inflate(R.layout.test_fragment,
                            container, false);
    }

    /*
     * Обработка событий жизненного цикла Фрагмента
    }
```

```
* -----
*/
@Override
public void onAttach (Context context)
{
    super.onAttach(context);
    Log.d(TAG, "----- onAttach -
Фрагмент прикрепляется к Активности");
}

@Override
public void onCreate (Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    Log.d(TAG, "----- onCreate -
Инициализация Фрагмента");
}

@Override
public void onActivityCreated (Bundle
        savedInstanceState)
{
    super.onActivityCreated(savedInstanceState);
    Log.d(TAG, "----- onActivityCreated -
Активность " + "завершила обработку
своего события onCreate");
}

@Override
public void onViewStateRestored (Bundle
        savedInstanceState)
{
    super.onViewStateRestored(savedInstanceState);
    Log.d(TAG, "----- onViewStateRestored -
Состояния виджетов " +
"Фрагмента восстановлены");
}
```

```
@Override
public void onStart()
{
    super.onStart();
    Log.d(TAG, "----- onStart – Фрагмент
        становится видимым для пользователя");
}

@Override
public void onResume()
{
    super.onResume();
    Log.d(TAG, "----- onResume – Фрагмент
        становится " + "доступным для
        взаимодействия с пользователем");
}

@Override
public void onPause()
{
    super.onPause();
    Log.d(TAG, "----- onPause – Фрагмент больше " +
        "не взаимодействует с пользователем");
}

@Override
public void onStop()
{
    super.onStop();
    Log.d(TAG, "----- onStop – Фрагмент более
        не виден пользователю");
}

@Override
public void onDestoryView()
{
    super.onDestoryView();
```

```
    Log.d(TAG, "----- onDestroyView – Фрагменту  
    необходимо " + "освободить ресурсы,  
    связанные с его виджетами");  
}  
  
@Override  
public void onDestroy()  
{  
    super.onDestroy();  
    Log.d(TAG, "----- onDestroy – Необходимо  
    выполнить " + "действия по окончательному  
    завершению работы Фрагмента");  
}  
  
@Override  
public void onDetach()  
{  
    super.onDetach();  
    Log.d(TAG,  
        "----- onDetach – Сейчас Фрагмент будет " +  
        "откреплен от Активности");  
}  
}
```

Как видно из Листинга 1.9, каждый метод обработчик события жизненного цикла Фрагмента вывод в Лог приложения подробное сообщение о своем запуске. Аналогично поступим и с классом Активности нашего приложения — переопределим все методы обработчики событий жизненного цикла Активности и в каждом таком методе осуществим вывод подробного сообщения в Лог приложения. Код методов (класс **MainActivity** из файла `/java/com.itstep.org/MainActivity.java`) приведен в Листинге 1.10.

**Листинг 1.10.** Методы–обработчики событий жизненного цикла Активности с выводом информации в Лог приложения

```
public class MainActivity extends AppCompatActivity
{
    //--- Class constants -----
    private final static String TAG = "##### MainActivity";

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

        //--- Вывод в Лог информации о запуске метода
        //--- onCreate -----
        Log.d(TAG, "----- onCreate – Активность создается");
    }

    /*
     * Обработка событий жизненного цикла Активности
     * -----
     */
    @Override
    protected void onStart()
    {
        super.onStart();
        Log.d(TAG,
            "-- onStart – Активность становится " +
            "видимой для пользователя");
    }

    @Override
    protected void onResume()
    {
```

```
super.onResume();
Log.d(TAG, "----- onResume – Активность
           выходит на передний план");
}

@Override
protected void onPause()
{
    super.onPause();
    Log.d(TAG, "----- onPause – Активность уходит
               с переднего плана");
}

@Override
protected void onStop()
{
    super.onStop();
    Log.d(TAG, "----- onStop – Активность перестает
               быть видимой");
}

@Override
protected void onDestroy()
{
    super.onDestroy();
    Log.d(TAG, "----- onDestroy –
               Активность уничтожается");
}
```

Теперь встроим Фрагмент в Активность. Для этого воспользуемся способом объявления Фрагмента в xml файле макета Активности (файл /res/layout/activity\_main.xml, см. Листинг 1.11).

## Листинг 1.11. Объявление Фрагмента из примера текущего раздела в файле макета Активности

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout

    ...
    android:orientation="vertical"
    tools:context="com.itstep.myapp.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:textSize="10pt"
        android:textColor="#003366"
        android:text="Модуль app. Фрагменты." />
    ...

    <Space
        android:layout_width="match_parent"
        android:layout_height="10dp"/>

    <fragment
        android:name="com.itstep.myapp.TestFragment"
        android:id="@+id/testFragment"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        tools:layout="@layout/test_fragment"/>

</LinearLayout>
```

Текущий пример практически не изменил внешний вид приложения (см. Рис. 1.3) и поэтому отдельного скриншота для данного примера не будет — суть примера в Логах приложения.

Давайте запустим наш пример и посмотрим на вывод Логов приложения. При запуске приложения мы увидим следующую последовательность сообщений, которая представлена в Листинге 1.12.

### Листинг 1.12. Логи приложения от методов обработчиков событий жизненного цикла Фрагмента и Активности

```
===== TestFragment: ----- onAttach -
Фрагмент прикрепляется к Активности
===== TestFragment: ----- onCreate -
Инициализация Фрагмента
===== TestFragment: ----- onCreateView -
Создание набора виджетов Фрагмента
##### MainActivity: ----- onCreate -
Активность создается
===== TestFragment: ----- onActivityCreated -
Активность завершила обработку своего
события onCreate
===== TestFragment: ----- onViewStateRestored -
Состояния виджетов Фрагмента
восстановлены
##### MainActivity: ----- onStart -
Активность становится видимой для пользователя
===== TestFragment: ----- onStart -
Фрагмент становится видимым для пользователя
##### MainActivity: ----- onResume -
Активность выходит на передний план
===== TestFragment: ----- onResume -
Фрагмент становится доступным для
взаимодействия с пользователем
```

В Листинге 1.12 для удобства жирным шрифтом выделены сообщения от методов жизненного цикла Активности. Как видим, Фрагмент создается (**onAttach**, **onCreate**, **onCreateView**) раньше Активности, но становится видимым

(**onStart**) и доступным для пользователя (**onResume**) позже Активности.

Теперь повернем устройство и посмотрим на сообщения от методов жизненного цикла. Эти сообщения приведены в Листинге 1.13.

### Листинг 1.13. Последовательность вызовов методов жизненного цикла Фрагмента и Активности

Уничтожение Активности:

```
===== TestFragment: ----- onPause – Фрагмент больше  
не взаимодействует с пользователем  
##### MainActivity: ----- onPause –  
Активность уходит с переднего плана  
===== TestFragment: ----- onStop –  
Фрагмент более не виден пользователю  
##### MainActivity: ----- onStop –  
Активность перестает быть видимой  
===== TestFragment: ----- onDestoryView –  
Фрагменту необходимо освободить ресурсы,  
связанные с его виджетами  
===== TestFragment: ----- onDestory –  
Необходимо выполнить действия по  
окончательному завершению работы Фрагмента  
===== TestFragment: ----- onDetach –  
Сейчас Фрагмент будет откреплен от Активности  
##### MainActivity: ----- onDestory –  
Активность уничтожается
```

Создание Активности:

```
===== TestFragment: ----- onAttach –  
Фрагмент прикрепляется к Активности  
===== TestFragment: ----- onCreate –  
Инициализация Фрагмента  
===== TestFragment: ----- onCreateView –  
Создание набора виджетов Фрагмента
```

```

##### MainActivity: ----- onCreate -
Активность создается
===== TestFragment: ----- onActivityCreated -
    Активность завершила обработку своего события
    onCreate
===== TestFragment: ----- onViewStateRestored -
    Состояния виджетов Фрагмента восстановлены
##### MainActivity: ----- onStart -
Активность становится видимой для пользователя
===== TestFragment: ----- onStart -
    Фрагмент становится видимым для пользователя
##### MainActivity: ----- onResume -
Активность выходит на передний план
===== TestFragment: ----- onResume -
    Фрагмент становится доступным для
    взаимодействия с пользователем

```

В Листинге 1.13 для удобства жирным шрифтом выделены сообщения от методов жизненного цикла Активности. Как видим, при уходе приложения с переднего плана (и при уничтожении приложения) события жизненного цикла Фрагмента отрабатывают перед соответствующими событиями жизненного цикла Активности. Затем происходит создание Активности и последовательность событий повторяется, как и в Листинге 1.12.

Если теперь нажать на кнопку «Назад» на устройстве (или на эмуляторе), то последовательность вызовов методов будет такая же, как и в первой части Листинга 1.13 (Уничтожение Активности).

Исходный код данного примера находится в модуле «app» среди файлов исходных кодов, которые прилагаются к данному уроку.

## 1.5. Методы класса android.app.FragmentManager

В этом разделе мы рассмотрим методы класса [android.app.FragmentManager](#), которые необходимо знать при взаимодействии с Фрагментами и со «Стеком Перехода Назад» для Фрагментов. Вот эти методы:

- **void addOnBackStackChangedListener(FragmentManager.OnBackStackChangedListener listener)** — метод позволяет установить обработчик событий изменения содержимого «Стека Переходов Назад». Принимает ссылку на объект, реализующий интерфейс [android.app.FragmentManager.OnBackStackChangedListener](#), в котором объявлен всего один метод **void onBackStackChanged()**.
- **Fragment findFragmentById(int id)** — метод находит в текущей Активности Фрагмент, идентификатор которого задан в параметре **id**.
- **Fragment findFragmentByTag(String tag)** — метод находит в текущей Активности Фрагмент по значению строкового идентификатора **tag** (Значение **tag** задается при добавлении Фрагмента в Активность с помощью методов **add**, **replace** объекта **android.app.FragmentTransaction**, о которых рассказывалось ранее в этом уроке).
- **int getBackStackEntryCount()** — метод возвращает количество элементов в «Стеке Переходов Назад».
- **void popBackStack()** — возвращает Фрагмент из «Стека Переходов Назад» на Активность и делает этот Фрагмент активным. При этом предыдущий активный Фрагмент уничтожается.
- **void popBackStack(int id, int flags)** — удаляет из «Стека Переходов Назад» все Фрагменты до Фрагмента

с идентификатором транзакции **id**, который помещается на Активность и становится активным. При этом предыдущий активный Фрагмент уничтожается.

- **void popBackStack(String name, int flags)** — удаляет из «Стека Переходов Назад» все Фрагменты до Фрагмента с названием транзакции **name**, который помещается на Активность и становится активным. При этом предыдущий активный Фрагмент уничтожается.
- **void putFragment(Bundle bundle, String key, Fragment fragment)** — метод позволяет разместить Фрагмент **fragment** в объекте **bundle** (`android.os.Bundle`). При размещении Фрагмента в **bundle** ему назначается строковый ключ **key**, который будет использован при извлечении Фрагмента из **bundle**.
- **Fragment getFragment(Bundle bundle, String key)** — метод позволяет извлечь из объекта `android.os.Bundle` (параметр **bundle**) ранее размещенный в нем Фрагмент с ключом **key**.

## 1.6. «Стек Переходов Назад» (Back Stack) для Фрагментов

Ранее в наших уроках никогда не затрагивался такой момент, что Android приложение можно считать коллекцией Активностей. И вот настало время вам об этом рассказать! При запуске приложения мы видим Главную Активность приложения (Активность, которая отображается при старте). Но главная (или первичная) Активность может запускать другие Активности. Например нашему приложению необходимо выполнить отправку электронной почты. Для этого запускается Активность другого

приложения, с помощью которого пользователь вводит все необходимые данные для отправки электронной почты, отправляет письмо и возвращается на предыдущую Активность. Так вот, когда в приложении запускается другая Активность, она сразу же помещается в «Стек Переходов Назад» поверх запущившей ее Активности (помните, что приложение это коллекция Активностей?). Причем Активности помещаются в «Стек Переходов Назад» в том порядке, в котором они открывались и из «Стека Переходов Назад» Активности извлекаются при нажатии на кнопку «Назад» в обратном порядке — то есть работает принцип обычного стека «последним зашел, первым вышел».

Рассмотрим более детально, как работает «Стек переходов Назад». Когда текущая Активность запускает другую Активность, новая Активность помещается в вершину «Стека Переходов Назад» и получает фокус. Предыдущая Активность остается в «Стеке Переходов Назад», но ее выполнение останавливается. Когда Активность останавливается, Операционная Система сохраняет текущее состояние ее пользовательского интерфейса. Когда пользователь нажимает кнопку «Назад», текущая Активность удаляется из вершины «Стека Переходов Назад» (то есть уничтожается) и возобновляется работа предыдущей Активности (восстанавливается предыдущее состояние ее пользовательского интерфейса). Активность добавляется в «Стек Переходов Назад» когда запускается другой Активностью и удаляется из «Стека Переходов Назад» когда пользователь нажимает кнопку «Назад».

Также необходимо добавить, что для всех приложений начальным «местом» является главный экран устройства,

то есть, при нажатии на кнопку «Назад» для первичной Активности, пользователь попадает на экран устройства, с которого он запускал приложение. То есть, главный экран устройства помещается в «Стек Переходов Назад» первым.

Аналогичная ситуация существует и для Фрагментов в Активности, только Фрагменты не добавляются автоматически в «Стек переходов Назад». Для добавления Фрагментов в «Стек Переходов Назад» необходимо вызывать метод **addToBackStack** объекта **android.app.FragmentManager**. В этом разделе мы с вами научимся работать со «Стеком Переходов Назад» для Фрагментов и нам пригодятся знания методов класса **android.app.FragmentManager**, с которыми мы познакомились в предыдущем разделе.

Для примера текущего раздела создадим модуль «app2», в который перенесем Фрагменты **FirstFragment** (файлы /res/layout/first\_fragment.xml и /java/com.itstep.myapp/FirstFragment.java, которые представлены в Листингах 1.1 и 1.2 соответственно) и **CalcFragment** (файлы /res/layout/calc\_fragment.xml и /java/com.itstep.myapp/CalcFragment.java, которые представлены в Листингах 1.4 и 1.5 соответственно) из предыдущего модуля «app». Эти Фрагменты и будут использованы нами для взаимодействия со «Стеком Переходов Назад».

В xml макете Активности разместим контейнер **android.widget.FrameLayout** (идентификатор **flFragmentContainer**), в который будут встраиваться Фрагменты **CalcFragment** и **FirstFragment**. Так же на Активности разместим кнопки «Добавить FirstFragment» (идентификатор **btnFirstFragment**) и «Добавить CalcFragment»

идентификатор **btnCalcFragment**), по нажатию на которые будут создаваться соответствующие Фрагменты и встраиваться в контейнер **flFragmentContainer**. При этом, пользователю будет доступна возможность возвращать Фрагменты из «Стека Переходов Назад» по нажатию на кнопку «Назад» устройства (или эмулятора). Xml код макета Активности (файл /res/layout/activity\_main.xml) представлен в Листинге 1.14.

**Листинг 1.14.** Xml верстка макета пользовательского интерфейса Активности для рассматриваемого в данном разделе примера

```
<LinearLayout  
    xmlns:android=  
        "http://schemas.android.com/apk/res/android"  
    xmlns:tools="http://schemas.android.com/tools"  
  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
  
    android:orientation="vertical"  
    tools:context="com.itstep.myapp2.MainActivity">  
  
    <TextView  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:layout_gravity="center_horizontal"  
        android:textSize="10pt"  
        android:textColor="#003366"  
        android:text="Модуль app2. Фрагменты." />  
  
    <Space  
        android:layout_width="match_parent"  
        android:layout_height="20dp"/>
```

```
<Button  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id	btnFirstFragment"  
    android:textAllCaps="false"  
    android:text="Добавить FirstFragment"  
    android:onClick="btnAddClick" />  
  
<Button  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id	btnCalcFragment"  
    android:textAllCaps="false"  
    android:text="Добавить CalcFragment"  
    android:onClick="btnAddClick" />  
  
<Space  
    android:layout_width="match_parent"  
    android:layout_height="20dp"/>  
  
<FrameLayout  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id/flFragmentContainer">  
</FrameLayout>  
  
</LinearLayout>
```

Внешний вид Активности из Листинга 1.14 изображен на Рис. 1.4.

Фрагменты встраиваться в Активность будут программным способом, для этого, как видно из Листинга 1.14, кнопкам с идентификаторами **btnFirstFragment** и **btnCalcFragment** назначен обработчик события нажатия — метод **btnAddClick**. В этом методе будут создаваться



**Рис. 1.4.** Внешний вид Активности из Листинга 1.14

требуемые Фрагменты и с помощью транзакции (объекта `android.app.FragmentTransaction`) будут встраиваться в контейнер `flFragmentContainer`. Кроме этого — и это очень важно — изменения Фрагментов на Активности, производимые этой транзакцией, будут размещаться в «Стек Переходов Назад» с помощью вызова метода `addToBackStack`. Код метода `btnAddClick` (и всего класса `MainActivity`) представлен в Листинге 1.15.

**Листинг 1.15.** Код класса `MainActivity`, демонстрирующий работу со «Стеком Переходов Назад» для Фрагментов

```
public class MainActivity extends AppCompatActivity
{
    //--- Class methods -----
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
```

```
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
}

public void btnAddClick(View v)
{
    Fragment F;

    switch (v.getId())
    {
        case R.id.btnFirstFragment:
            F = new FirstFragment();
            break;

        case R.id.btnCalcFragment:
            F = new CalcFragment();
            break;

        default:
            return;
    }

    //--- Получение ссылки на
    //--- android.app.FragmentManager -----
    FragmentManager FM = this.getFragmentManager();

    //--- Создание транзакции для изменения Фрагментов --
    FragmentTransaction FT = FM.beginTransaction();

    //--- Добавление нового Фрагмента в Активность ---
    FT.replace(R.id.flFragmentContainer, F);
    FT.addToBackStack(null);
    FT.commit();
}

@Override
public void onBackPressed()
```

```
{  
    FragmentManager FM = this.getFragmentManager();  
    FM.popBackStack();  
}  
}
```

Итак, еще раз обратим внимание на то, что для размещения изменений производимых текущей транзакцией (объект `android.app.FragmentTransaction`) в «Стек Переходов Назад» необходимо воспользоваться методом `addToBackStack` (в Листинге 1.15 вызов метода выделен жирным шрифтом).

И еще один важный момент! По умолчанию, нажатие на кнопку «Назад» устройства приводит к возврату предыдущей Активности. Для того чтобы в нашем приложении нажатие на кнопку «Назад» приводило к возврату из «Стека Переходов Назад» предыдущего Фрагмента, необходимо для класса Активности нашего приложения переопределить метод `onBackPressed` (метод обратного вызова, вызывается при нажатии на кнопку «Назад»). И в переопределенном методе необходимо осуществить вызов метода `popBackStack` объекта `android.app.FragmentManager`. В Листинге 1.15 метод `onBackPressed` выделен жирным шрифтом.

Правда, такая реализация метода не является единственной — ведь фактически приложение лишает пользователя функциональности использовать кнопку «Назад» для возврата к предыдущей Активности или к главному экрану устройства. Поэтому, как вариант, возможна и такая реализация метода `onBackPressed`, как в Листинге 1.16:

**Листинг 1.16.** Вариант реализации метода `onBackPressed`

с вызовом родительского метода для случая, если в «Стеке Переходов Назад» не осталось Фрагментов

```

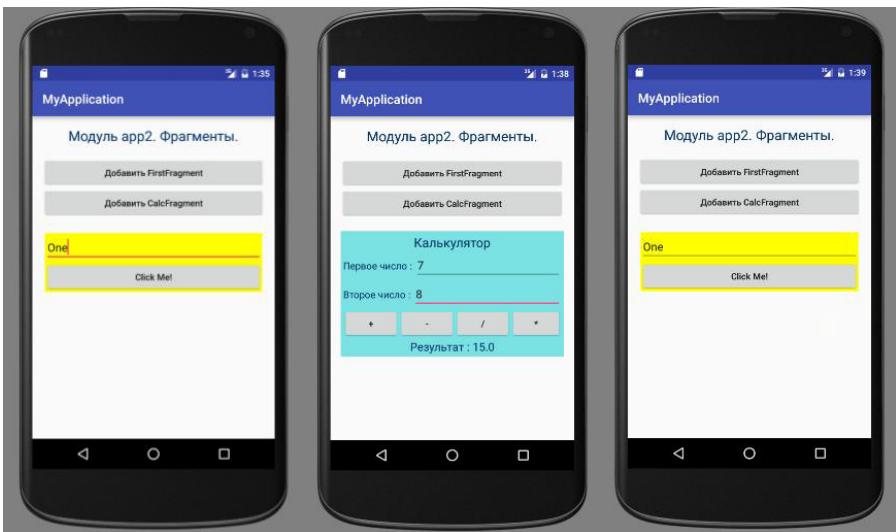
@Override
public void onBackPressed()
{
    FragmentManager FM = this.getFragmentManager();

    if (FM.getBackStackEntryCount() == 0)
    {
        //-- "Стек Переходов Назад" пуст – возвращаем
        //-- предыдущую Активность
        super.onBackPressed();
    }
    else
    {
        //-- Есть Фрагменты в стеке – возвращаем
        //-- предыдущий -----
        FM.popBackStack();
    }
}

```

Как видно из Листинга 1.16, в случае, если больше не осталось Фрагментов в «Стеке Переходов Назад», то осуществляется вызов родительского метода **onBackPressed**, с целью вернуть из «Стека Переходов Назад» предыдущую Активность или главный экран устройства. Внешний вид работы примера из Листингов 1.14, 1.15, 1.16 изображен на Рис. 1.5.

На Рис. 1.5 изображена следующая последовательность действий: пользователь нажимает на кнопку «Добавить FirstFragment» и в появившемся Фрагменте **FirsrFragment** вводит слово «One». Затем пользователь нажимает на



**Рис. 1.5.** Внешний вид работы примера текущего раздела  
(Листинги 1.14, 1.15, 1.16)

кнопку «Добавить CalcFragment» и в появившемся Фрагменте **CalcFragment** вводит числа 7 и 8 и нажимает на кнопку «+». Пользователь видит результат сложения чисел 7 и 8. Далее, пользователь нажимает на кнопку «Назад» устройства и видит, что из «Стека Переходов Назад» возвращается Фрагмент **FirstFragment** с введенным в текстовое поле словом «One». Проделайте такую же или аналогичную последовательность действий, чтобы наблюдать за работой приложения взаимодействующего со «Стеком Переходов Назад».

Полный код примера находится в модуле «app2» проекта в файлах исходных кодов, которые прилагаются к данному уроку.

## 1.7. Запуск Фрагмента без пользовательского интерфейса

Фрагменты можно использовать и без пользовательского интерфейса для решения каких-либо фоновых задач Активности (здесь понятие «фоновый» не имеет никакого отношения к многопоточности — все Фрагменты исполняются в том же потоке, что и Активность).

Для того, чтобы добавить Фрагмент без пользовательского интерфейса в Активность, необходимо воспользоваться методом объекта **android.app.FragmentTransaction**:

```
FragmentTransaction add (Fragment fragment, String tag);
```

или методом (**android.app.FragmentTransaction**):

```
FragmentTransaction add (int containerViewId,  
                         Fragment fragment, String tag);
```

только в качестве значения параметра **containerViewId** здесь необходимо передать значение 0 (ноль).

Фрагмент будет добавлен, но, поскольку для него не будет найден контейнер в Активности (идентификатор контейнера 0 — это несуществующий контейнер), то набор виджетов Фрагмента не будет прикреплен к виджетам Активности. Как следствие — в реализации метода **onCreateView** для Фрагмента нет необходимости. В описании API на [сайте разработчиков](#) говорится, что в таком случае не будет вызван метод обратного вызова **onCreateView**. Однако, проверено, этот метод вызывается — поэтому, при желании, этот метод можно реализовать и вернуть, например значение **null** (что и будет сделано в нашем примере).

Также, на [сайте разработчиков](#) говорится, что можно использовать «невидимый» Фрагмент в качестве фонового потока. Поясним, что даже невидимые Фрагменты (то есть Фрагменты без пользовательского интерфейса) исполняются в том же потоке, что и Активность. Фоновый поток для невидимого Фрагмента нужно будет создавать и запускать самостоятельно, если есть в этом необходимость. Без дополнительных вторичных потоков невидимый Фрагмент будет получать только события своего жизненного цикла.

Важно обратить внимание на следующий факт! Если у фрагмента нет пользовательского интерфейса, то **tag** (строковый идентификатор) является единственным способом идентификации (нахождения) Фрагмента в Активности. Для того, чтобы найти Фрагмент в Активности, необходимо будет вызвать метод **findFragmentByTag** объекта **android.app.FragmentManager** (об этом методе уже рассказывалось в данном уроке).

Рассмотрим пример создания Фрагмента без пользовательского интерфейса. В модуль «app2» добавим класс для нашего «невидимого» Фрагмента. Класс будет называться **InvisibleFragment** и будет находиться в файле `/java/com.itstep.myapp2/InvisibleFragment.java`. Класс **InvisibleFragment** (неполностью) приведен в Листинге 1.17.

#### Листинг 1.17. Код класса InvisibleFragment для Фрагмента без пользовательского интерфейса

```
package com.itstep.myapp2;
import ...
/**
 * Class InvisibleFragment – Фрагмент без
 * пользовательского интерфейса -----
 */
```

```
public class InvisibleFragment extends Fragment
{
    //--- Class constants -----
    private final static String TAG =
        "===== InvisibleFragment";

    //--- Class methods -----
    @Override
    public View onCreateView(LayoutInflater inflater,
                            ViewGroup container,
                            Bundle savedInstanceState)
    {
        Log.d(TAG, "----- onCreateView");
        return null;
    }

    /*
     * Обработка событий жизненного цикла Фрагмента
     * -----
     */
    @Override
    public void onAttach (Context context)
    {
        super.onAttach(context);
        Log.d(TAG, "----- onAttach -
Фрагмент прикрепляется к Активности");
    }

    ...
    @Override
    public void onDetach()
    {
        super.onDetach();
        Log.d(TAG, "----- onDetach - Сейчас Фрагмент
будет откреплен от Активности");
    }
}
```

Класс **InvisibleFragment** в Листинге 1.17 приведен неполностью потому, что код этого класса практически ничем не отличается от класса **TestFragment**, код которого приведен в Листинге 1.9. Классы **InvisibleFragment** и **TestFragment** отличаются друг от друга только реализацией метода **onCreateView** (в классе **InvisibleFragment** возвращается значение `null` — за ненадобностью пользовательского интерфейса).

Далее, в макет Активности (файл `/res/layout/activity_main.xml`) добавим две кнопки «Добавить InvisibleFragment» (идентификатор `btnAddInvFrgament`) и «Удалить InvisibleFragment» (идентификатор `btnDelInvFrgament`). Часть кода макета Активности с xml версткой этих кнопок представлена в Листинге 1.18.

**Листинг 1.18.** Кнопки в макете Активности для добавления и удаления Фрагмента без пользовательского интерфейса

```
<Button  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id(btnAddInvFrgament"  
    android:textAllCaps="false"  
    android:text="Добавить InvisibleFragment"  
    android:onClick="btnInvClick" />  
  
<Button  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id(btnDelInvFrgament"  
    android:textAllCaps="false"  
    android:text="Удалить InvisibleFragment"  
    android:onClick="btnInvClick" />
```

Кнопкам «Добавить InvisibleFragment» и «Удалить InvisibleFragment» назначен обработчик события — метод **btnInvClick**. Который работает следующим образом:

- При нажатии на кнопку «Добавить InvisibleFragment» будет сначала осуществлена проверка, не добавлен ли уже в Активность Фрагмент с тегом «Invisible»? (Именно такое значение тега будем присваивать невидимому Фрагменту). Если такой Фрагмент уже добавлен в Активность, то повторное его добавление не будет осуществлено и выведется сообщение об ошибке в Тосте — «Невидимый Фрагмент уже добавлен!», в противном случае, Фрагмент будет добавлен с помощью метода **add**.
- При нажатии на кнопку «Удалить InvisibleFragment» будет сначала осуществлена проверка, существует в Активности Фрагмент с тегом «Invisible»? Если не существует, то будет выведено сообщение в Тосте — «Невидимый Фрагмент уже удален!», в противном случае, Фрагмент будет удален из Активности с помощью метода **remove**.

Программный код метода **btnInvClick** приведен в Листинге 1.19.

**Листинг 1.19.** Код метода обработчика **btnInvClick** для обработки событий нажатия на кнопки добавления и удаления невидимого Фрагмента

```
public void btnInvClick (View v)
{
    //-- Получаем ссылку на объект android.app.
    //-- FragmentManager -----
    FragmentManager FM = this.getFragmentManager();
```

```
//-- Находим "невидимый" Фрагмент на Активности --
Fragment F = FM.findFragmentByTag("Invisible");

//-- Начинаем транзакцию -----
FragmentTransaction FT = FM.beginTransaction();

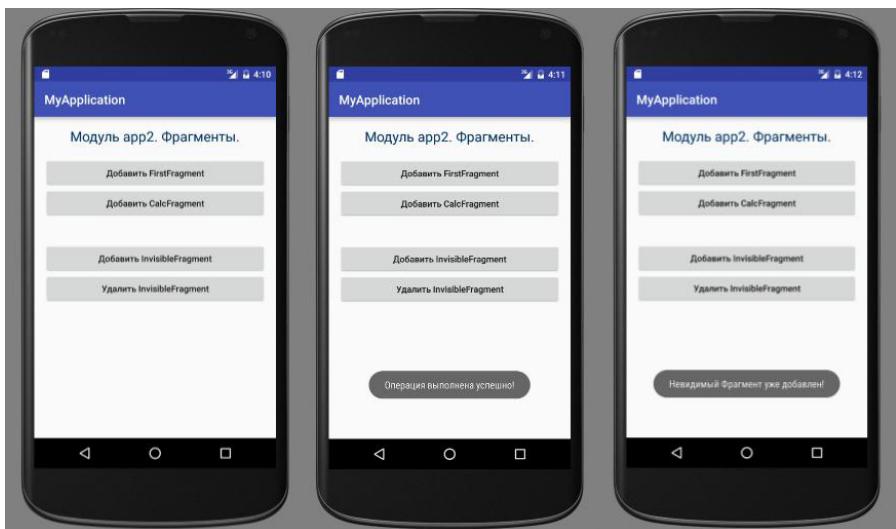
//-- Выполняем действие в зависимости от нажатой
//-- кнопки -----
switch (v.getId())
{
//-- Кнопка "Добавить Невидимый Фрагмент" -----
    case R.id.btnAddInvFragment:
        if (F != null)
        {
            Toast.makeText(this,
                "Невидимый Фрагмент уже добавлен!",
                Toast.LENGTH_LONG).show();
            break;
        }
        F = new InvisibleFragment();
        FT.add(F, "Invisible");
        break;

//-- Кнопка "Удалить невидимый Фрагмент" -----
    case R.id.btnDeleteInvFragment:
        if (F == null)
        {
            Toast.makeText(this,
                "Невидимый Фрагмент уже удален!",
                Toast.LENGTH_LONG).show();
            break;
        }
        FT.remove(F);
        break;
}

//-- Подтверждаем Транзакцию, если необходимо ---
if (!FT.isEmpty())
```

```
{  
    FT.commit();  
    Toast.makeText(this,  
        "Операция выполнена успешно!",  
        Toast.LENGTH_LONG).show();  
}  
}  
}
```

Внешний вид работы примера из данного раздела изображен на Рис. 1.6.



**Рис. 1.6.** Внешний вид работы приложения из рассматриваемого в данном разделе примера на добавление и удаление Фрагмента без пользовательского интерфейса

Последовательность действий, изображенная на Рис. 1.6 следующая. Пользователь нажимает на кнопку «Добавить InvisibleFragment». Приложение проверяет, что такой Фрагмент не добавлен, добавляет его и выводит сообщение «Операция выполнена успешно!». Затем пользователь

снова нажимает на кнопку «Добавить InvisibleFragment» и приложение снова проверяет, добавлен ли на Активность такой Фрагмент. Поскольку Фрагмент уже добавлен, то приложение выводит сообщение об ошибке «Невидимый Фрагмент уже добавлен». Разумеется, если теперь нажать на кнопку «Удалить InvisibleFragment», то Фрагмент будет успешно удален и приложение выведет сообщение «Операция выполнена успешно!». Попробуйте работу этого примера самостоятельно.

В завершении рассмотрения примера, обратим внимание на то, каким образом приложение определяет, добавлен ли невидимый Фрагмент «InvisibleFragment» с тегом «Invisible» в Активность или нет. Для этой цели используется метод поиска Фрагментов по тегу **findFragmentByTag** объекта класса **android.app.FragmentManager** (об этом методе уже упоминалось в данном уроке). В Листинге 1.19 этот фрагмент кода выделен жирным шрифтом. Если метод **findFragmentByTag** возвращает значение null, то Фрагмент с таким тегом не добавлен в Активность, в противном случае (значение не null) – Фрагмент в Активность будет добавлен.

Поскольку Фрагмент не имеет пользовательского интерфейса, то в рассматриваемом примере Фрагмент InvisibleFragment выводит сообщения о прохождении жизненного цикла в Лог приложения. Так, при добавлении Фрагмента мы увидим в Логах следующее:

```
===== InvisibleFragment: ----- onAttach  
===== InvisibleFragment: ----- onCreate  
===== InvisibleFragment: ----- onCreateView  
===== InvisibleFragment: ----- onActivityCreated  
===== InvisibleFragment: ----- onStart  
===== InvisibleFragment: ----- onResume
```

А при удалении невидимого Фрагмента, Лог приложения будет таким:

```
===== InvisibleFragment: ----- onPause
===== InvisibleFragment: ----- onStop
===== InvisibleFragment: ----- onDestoryView
===== InvisibleFragment: ----- onDestory
===== InvisibleFragment: ----- onDetach
```

Исходный код примера этого раздела находится в модуле «app2» среди файлов с исходными кодами, которые прилагаются к данному уроку.

## 1.8. Фрагмент «Диалоговое окно». Класс android.app.DialogFragment

В одном из прошлых уроков мы с вами знакомились, как использовать Диалоговые окна в Android приложениях с помощью класса [android.app.AlertDialog](#). Но класс [android.app.AlertDialog](#) является не единственным инструментом для создания Диалоговых окон. Другим способом является использование класса [android.app.DialogFragment](#), который, хоть и является Фрагментом, ведет себя как Диалоговое окно.

Иерархия классов для класса [android.app.DialogFragment](#) следующая:

```
java.lang.Object
  |
  +--- android.app.Fragment
      |
      +--- android.app.DialogFragment
```

То есть, класс [android.app.DialogFragment](#) является прямым наследником класса [android.app.Fragment](#).

Создать Диалоговое окно `android.app.DialogFragment` можно двумя способами:

- Первый способ уже хорошо нам знаком из этого урока — это способ создания Фрагмента с помощью создания производного класса от класса `android.app.Fragment` (а точнее от класса `android.app.DialogFragment`) с переопределением метода `onCreateView`, в котором с помощью объекта `android.view.LayoutInflater` создается набор виджетов пользовательского интерфейса Фрагмента, на основе специально созданного xml файла макета.
- Второй способ заключается в создании класса, производного от класса `android.app.DialogFragment`, и переопределении метода `onCreateDialog`, в котором с помощью объекта `android.app.AlertDialog.Builder` и будет создано Диалоговое окно.

Мы рассмотрим оба способа создания Диалоговых окон на основе класса `android.app.DialogFragment`. Для приложения, которое послужит примером для данного раздела, создан модуль «app3» среди файлов с исходными кодами, прилагаемыми к данному уроку.

Перед тем, как приступить к рассмотрению примера создания Диалоговых окон на основе Фрагментов, познакомимся с некоторыми важными методами класса `android.app.DialogFragment`:

- `void dismiss()` — закрывает Диалоговое окно (окно исчезает с экрана).
- `Dialog getDialog()` — возвращает ссылку на объект `android.app.Dialog`, который представляет данное Диалоговое окно.

- **void onDismiss(DialogInterface dialog)** — метод обратного вызова, вызывается при закрытии окна. Чтобы получить событие закрытия Диалогового окно, этот метод необходимо переопределить в производном классе.
- **void onCancel(DialogInterface dialog)** — метод обратного вызова, вызывается когда Диалоговое окно отменено (то есть пользователь кликнул за переделами Диалогового окна). Чтобы получить событие отмены Диалогового окна, этот метод необходимо переопределить в производном классе. Чтобы Диалоговое окно невозможно было отменить, необходимо воспользоваться методом **setCancelable(true)**.
- **void setCancelable(boolean cancelable)** — если передать значение true, Диалоговое окно невозможно будет отменить. В этом случае Диалоговое окно может быть закрыто только по нажатию на одну из кнопок Диалогового окна.
- **void setShowsDialog(boolean showsDialog)** — если передать значение true, то Фрагмент будет отображен как Диалоговое окно. Если передать значение false, то набор виджетов Фрагмента встраиваются в Активность, как для обычного Фрагмента.
- **void setStyle(int style, int theme)** — позволяет задать стиль внешнего вида Диалогового окна. Значения стиля заданы в виде констант: **STYLE\_NORMAL**, **STYLE\_NO\_TITLE**, **STYLE\_NO\_FRAME**, **STYLE\_NO\_INPUT** ([что это за стили см. в документации](#)).

- **void show(FragmentManager manager, String tag)** — отображает Диалоговое окно, используя объект **android.app.FragmentManager** (параметр **manager**). Этот метод сам создает транзакцию **android.app.FragmentTransaction** и сам ее подтверждает с помощью вызова метода **commit()**. Параметр **tag** задает строковый идентификатор Фрагмента.
- **int show(FragmentTransaction transaction, String tag)** — отображает Диалоговое окно используя объект транзакции **android.app.FragmentTransaction** (параметр **transaction**). Метод самостоятельно подтверждает транзакцию с помощью вызова метода **commit()**. Метод возвращает идентификатор транзакции, который вернул метод **commit()**. Параметр **tag** задает строковый идентификатор Фрагмента.
- **void setArguments(Bundle args)** — этот метод наследуется от класса **android.app.Fragment**. Метод предназначен для передачи набора параметров в Фрагмент. Набор параметров (значений, которые необходимы Фрагменту для работы) передается в объекте **android.os.Bundle** (параметр **args**). Этот метод необходимо вызвать перед тем, как Фрагмент будет прикреплен к Активности — т. е. метод рекомендуется вызывать сразу же после создания Фрагмента. Набор параметров, которые передаются с помощью этого метода, существует и доступен для Фрагмента на протяжении всего жизненного цикла до его уничтожения. Фрагмент может получить доступ к этому набору параметров с помощью вызова метода **getArguments()**.

- **Bundle getArguments ()** — этот метод наследуется от класса **android.app.Fragment**. Метод предназначен для получения набора параметров, который передан Фрагменту с помощью метода **setArguments(Bundle args)**. Метод чаще всего вызывается из методов самого Фрагмента.

Итак, перейдем к рассмотрению примера по созданию Диалоговых окон на основе класса **android.app.DialogFragment**. Как уже говорилось выше, будут рассмотрены оба варианта создания Диалоговых окон: с помощью переопределения метода **onCreateView()**, в котором будет использован xml файла макета для создания набора виджетов Фрагмента, и с помощью переопределения метода **onCreateDialog()**, в котором набор виджетов пользовательского интерфейса Фрагмента будет создан с помощью объекта **android.app.AlertDialog.Builder**.

Что будет представлять рассматриваемый пример с точки зрения функциональности для пользователя? Это будет приложение, в котором пользователю будет представлена возможность редактировать два набора значений — «Фамилию»—«Имя» и «Рост»—«Вес». Редактирование «Фамилии» и «Имени» будет осуществляться с помощью Фрагмента с переопределением метода **onCreateView()**. Для этого Фрагмента создан класс **DialogFragmentOne** (файл `/java/com.itstep.myapp3/DialogFragment.java`). Редактирование «Роста» и «Веса» будет осуществляться с помощью Фрагмента с переопределением метода **onCreateDialog()**. Для этого Фрагмента создан класс **DialogFragmentTwo** (файл `/java/com.itstep.myapp3/DialogFragmentTwo.java`).

Макет пользовательского интерфейса Активности (файл /res/layout/activity\_main.xml) приведен в Листинге 1.20.

**Листинг 1.20.** Xml макет Активности  
для примера с Диалоговыми окнами на основе  
класса android.app.DialogFragment

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.itstep.myapp3.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:textSize="10pt"
        android:textColor="#003366"
        android:text="Модуль app3. DialogFragment." />

    <Space
        android:layout_width="match_parent"
        android:layout_height="10dp" />

    <TableLayout
        android:background="#fafad1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="4dp"
        android:stretchColumns="*">
```

```
<TableRow>
    <TextView
        android:text="Фамилия: "
        android:layout_gravity=
            "center_vertical|right"
        android:textSize="8pt" />

    <TextView
        android:id="@+id/tvLastName"
        android:layout_gravity=
            "center_vertical|left"
        android:textSize="8pt" />
</TableRow>

<TableRow>
    <TextView
        android:text="Имя: "
        android:layout_gravity=
            "center_vertical|right"
        android:textSize="8pt" />

    <TextView
        android:id="@+id/tvFirstName"
        android:layout_gravity=
            "center_vertical|left"
        android:textSize="8pt" />
</TableRow>

<TableRow>
    <Button
        android:layout_span="2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textAllCaps="false"
        android:id="@+id/btnOne"
        android:onClick="btnClick"
```

```
        android:text="Диалог:  
        Ввести Фамилию и Имя"/>  
    </TableRow>  
  </TableLayout>  
  
<Space  
    android:layout_width="match_parent"  
    android:layout_height="10dp" />  
  
<TableLayout  
    android:background="#daf2cc"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:padding="4dp"  
  
    android:stretchColumns="*"  
  
<TableRow>  
    <TextView  
        android:text="Рост: "  
        android:layout_gravity=  
            "center_vertical|right"  
        android:textSize="8pt" />  
  
    <TextView  
        android:id="@+id/tvHeight"  
        android:layout_gravity=  
            "center_vertical|left"  
        android:textSize="8pt" />  
  </TableRow>  
  
<TableRow>  
    <TextView  
        android:text="Вес: "  
        android:layout_gravity=  
            "center_vertical|right"  
        android:textSize="8pt" />
```

```
<TextView  
        android:id="@+id/tvWeight"  
        android:layout_gravity=  
            "center_vertical|left"  
        android:textSize="8pt" />  
</TableRow>  
  
<TableRow>  
    <Button  
        android:layout_span="2"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:textAllCaps="false"  
        android:id="@+id	btnTwo"  
        android:onClick="btnClick"  
        android:text="Диалог: Ввести Рост и Вес" />  
  
</TableRow>  
</TableLayout>  
</LinearLayout>
```

Внешний вид Активности приложения из Листинга 1.20 изображен на Рис. 1.7.

Как видно из Рис. 1.7 и Листинга 1.20, для редактирования набора полей «Фамилия» и «Имя» предназначена кнопка «Диалог: Ввести Фамилию и Имя» (идентификатор кнопки **btnOne**). А для редактирования полей «Рост» и «Вес» предназначена кнопка «Диалог: Ввести Рост и Вес» (идентификатор кнопки **btnTwo**). Этим кнопкам назначен обработчик события — метод **btnClick()**. По нажатию на кнопку **btnOne** будет появляться Диалоговое окно на основе класса **DialogFragmentOne**, а по



**Рис. 1.7.** Внешний вид Активности из Листинга 1.20

нажатию на кнопку **btnTwo** будет появляться Диалоговое окно на основе класса Фрагмента **DialogFragmentTwo**. Причем, пользователю предоставляется возможность с помощью Диалоговых окон редактировать значения, которые находятся в текстовых полях (идентификаторы **tvLastName**, **tvfirstName** для «Фамилия» и «Имя» соответственно, и идентификаторы **tvHeight**, **tvWeight** для «Рост» и «Вес» соответственно) на Активности. Таким образом, существует необходимость обмена значениями (с помощью методов **setArguments()** и **getArguments()**) между Активностью и Фрагментами, что также рассматривается в примере.

Код метода обработчика событий нажатия **btnClick** класса **MainActivity** приведен в Листинге 1.21.

**Листинг 1.21.** Метод btnClick обработчика событий нажатий на кнопки создания Диалоговых окон на основе Фрагментов

```
public class MainActivity extends AppCompatActivity
{
    //--- Class methods -----
    @Override
    protected void onCreate(Bundle
                           savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void btnClick(View v)
    {
        //--- Получение ссылки на объект
        //--- android.app.FragmentManager -----
        FragmentManager FM = this.getFragmentManager();

        switch (v.getId())
        {
            /*
             * Создание Диалогового окна DialogFragmentOne
             * -----
             */
            case R.id.btnOne:
                DialogFragmentOne F =
                    new DialogFragmentOne();

                //--- Создаем параметры для передачи их
                //--- в Диалоговое окно -----
                Bundle bundleArgs = new Bundle();
                bundleArgs.putInt(
                    DialogFragmentOne.DIALOGONE_LASTNAME,
                    R.id.tvLastName);
        }
    }
}
```

```
        bundleArgs.putInt(
            DialogFragmentOne.
            DIALOGONE_FIRSTNAME,
            R.id.tvFirstName);
        F.setArguments(bundleArgs);

    //--- Отображаем Диалоговое окно -----
        F.show(FM, "DialogOne");
        break;

/*
 * Создание Диалогового окна DialogFragmentTwo
 * -----
 */
case R.id.btnExit:
    DialogFragmentTwo F2 =
        new DialogFragmentTwo();

//--- Создаем параметры для передачи их
//--- в Диалоговое окно -----
    Bundle bundleArgs2 = new Bundle();

    bundleArgs2.putInt(
        DialogFragmentTwo.DIALOGTWO_HEIGHT,
        R.id.tvHeight);

    bundleArgs2.putInt(
        DialogFragmentTwo.DIALOGTWO_WEIGHT,
        R.id.tvWeight);

    F2.setArguments(bundleArgs2);

//--- Отображаем Диалоговое окно -----
    F2.show(FM, "DialogTwo");
    break;
}
}
```

Как видно из Листинга 1.21 (первый выделенный жирным шрифтом фрагмент кода), в Диалоговое окно, создаваемое на основе класса **DialogFragmentOne** передаются идентификаторы текстовых полей Активности (**tvLastName** и **tvFirstName**), из которых Фрагмент прочитает значения «Фамилия» и «Имя» и отобразит их для редактирования в Диалоговом окне **DialogFragmentOne**. После закрытия Диалогового окна **DialogFragmentOne**, новые значения для «Фамилия» и «Имя», которые ввел пользователь, будут помещены в эти же текстовые поля на Активности. Как работает Диалоговое окно **DialogFragmentOne** изображено на Рис. 1.8.

Код класса **DialogFragmentOne** приведен в Листинге 1.22.

### Листинг 1.22. Код класса Фрагмента DialogFragmentOne

```
package com.itstep.myapp3;

import ...

/**
 * Class DialogFragmentOne – Пример создания
 * Диалогового окна с помощью xml файла макета
 * и переопределения метода onCreateView
 *
 */
public class DialogFragmentOne extends DialogFragment
    implements View.OnClickListener
{
    //--- Class constants -----
    /**
     * Ключ для Bundle параметра, который содержит
     * идентификатор текстового поля TextView, из
```

```
* которого Диалоговое окно считывает
* Фамилию. После закрытия Диалогового окна
* по кнопке "Применить" в это текстовое поле
* будет записана Фамилия, которую ввел пользователь
*/
public final static String DIALOGONE_LASTNAME =
        "dialog_one_lastname";

/**
 * Ключ для Bundle параметра, который содержит
 * идентификатор текстового поля TextView, из
 * которого Диалоговое окно считывает Имя.
 * После закрытия Диалогового окна по кнопке
 * "Применить" в это текстовое поле будет
 * записано Имя, которое ввел пользователь
*/
public final static String DIALOGONE_FIRSTNAME =
        "dialog_one_firstname";

//--- Class members -----
/***
 * Ссылка на текстовое поле EditText, в которое
 * пользователь будет вводить Фамилию
*/
private EditText edtLastName;

/***
 * Ссылка на текстовое поле EditText, в которое
 * пользователь будет вводить Имя
*/
private EditText edtFirstName;

//--- Class methods -----
@Override
public View onCreateView(LayoutInflater inflater,
                        ViewGroup containerId,
                        Bundle savedInstanceState)
```

```
{  
    //--- Создаем набор виджетов пользовательского  
    //--- интерфейса Фрагмента -----  
    View view = inflater.inflate(  
        R.layout.dialog_one_fragment,  
        containerViewId, false);  
  
    //--- Назначение обработчиков событий для кнопок  
    //--- "Apply" и "Cancel" -----  
    view.findViewById(R.id.btnApply).  
        setOnClickListener(this);  
    view.findViewById(R.id.btnCancel).  
        setOnClickListener(this);  
  
    //--- Инициализация полей объекта -----  
    this.edtLastName = (EditText) view.  
        findViewById(R.id.edtLastName);  
    this.edtFirstName = (EditText) view.  
        findViewById(R.id.edtFirstName);  
    return view;  
}  
  
@Override  
  
public void onResume()  
{  
    super.onResume();  
  
    //--- Получение Bundle-переметров -----  
    Bundle bundleArgs = this.getArguments();  
  
    //--- Инициализация Текстовых полей EditText  
    //--- Диалогового окна -----  
  
    if (bundleArgs.containsKey(DialogFragmentOne.  
        DIALOGONE_LASTNAME))  
    {
```

```
        this.edtLastName.setText(
            ((TextView) this.getActivity() .
            findViewById(bundleArgs.getInt(
                DialogFragmentOne.DIALOGONE_LASTNAME))).
            getText());
    }

    if (bundleArgs.containsKey(DialogFragmentOne.
        DIALOGONE_FIRSTNAME))
    {
        this.edtFirstName.setText(
            ((TextView) this.getActivity() .
            findViewById(bundleArgs.getInt(
                DialogFragmentOne.
                DIALOGONE_FIRSTNAME))).getText());
    }
}

@Override
public void onClick(View v)
{
    switch (v.getId())
    {
/*
 * Нажатие на кнопку "Применить" -----
 */
        case R.id.btnApply:
//-- Помещаем введенные пользователем
//-- Фамилию и Имя в текстовые поля Активности --
        Bundle bundleArgs = this.getArguments();
        if (bundleArgs.containsKey(
            DialogFragmentOne.DIALOGONE_LASTNAME))
        {
            ((TextView) this.getActivity() .
            findViewById(bundleArgs.getInt(
                DialogFragmentOne.DIALOGONE_LASTNAME))).
            setText(this.edtLastName.getText());
        }
    }
}
```

```
    if (bundleArgs.containsKey(
        DialogFragmentOne.DIALOGONE_FIRSTNAME))
    {
        ((TextView) this.getActivity().findViewById(bundleArgs.getInt(
            DialogFragmentOne.
            DIALOGONE_FIRSTNAME))).setText(this.edtFirstName.getText());
    }

    //-- Закрываем Диалоговое окно -----
    this.dismiss();
    break;

/*
 * Нажатие на кнопку "Отменить" -----
 */
    case R.id.btnCancel:
    //-- Закрываем Диалоговое окно -----
        this.dismiss();
        break;
    }
}
```

Внешний вид работы Диалогового окна класса **DialogFragmentOne** изображен на Рис. 1.8.

В методе `onCreateView` класса `DialogFragmentOne` (Листинг 1.22) создание набора виджетов пользовательского интерфейса для Диалогового окна происходит аналогичным способом, как и для рассматриваемых ранее в этом уроке Фрагментов — с помощью объекта `android.view.LayoutInflater` и xml файла макета (`/res/layout/dialog_one_fragment.xml`). Содержимое xml файла приведено в Листинге 1.23.

**Листинг 1.23.** Xml файл макета файла  
/res/layout/dialog\_one\_fragment.xml для пользовательского  
интерфейса Диалогового окна DialogFragmentOne

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"

    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="4dp"
    android:stretchColumns="*">

    <TableRow>
        <TextView
            android:text="Фамилия"
            android:layout_gravity="center"
            android:textSize="8pt" />

        <EditText android:id="@+id/edtLastName" />
    </TableRow>

    <TableRow>
        <TextView
            android:text="Имя"
            android:layout_gravity="center"
            android:textSize="8pt" />

        <EditText android:id="@+id/edtFirstName" />
    </TableRow>

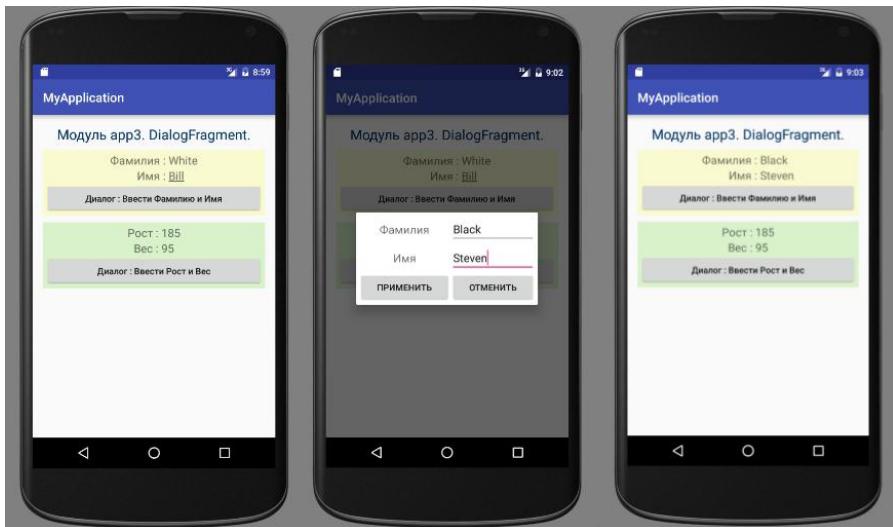
    <TableRow>
        <Button
            android:text="Применить"
            android:id="@+id/btnApply" />
    
```

```

<Button
    android:text="Отменить"
    android:id="@+id btnCancel" />
</TableRow>

</TableLayout>

```



**Рис. 1.8.** Внешний вид работы с Диалоговым окном, построенном на основе класса DialogFragmentOne

Обратим внимание на следующий момент. Диалоговое окно **DialogFragmentOne** получает в качестве параметров (набор параметров в пакете **android.os.Bundle**) идентификаторы текстовых полей Активности (**tvLastName**, **tvFirstName**), из которых необходимо прочитать значения для «Фамилия» и «Имя», чтобы затем эти значения разместить в Диалоговом окне (в полях **android.widget.EditText** с идентификаторами **edtLastName** и **edtFirstName** соответственно). Так вот, Фрагмент не может обратиться

к текстовым полям Активности в методе `onCreateView`. Почему? — Потому что он еще не прикреплен к Активности. Следовательно, для обращения к текстовым полям Активности лучше всего подойдет обработчик события жизненного цикла, который запускается после прикрепления Фрагмента к Активности. В классе `DialogFragmentOne` для этих целей выбран обработчик события `onResume()`.

При закрытии окна по кнопке «Применить» значения текстовых полей `android.widget.EditText` Диалогового окна «Фамилия» и «Имя» (идентификаторы `edtLastName` и `edtFirstName` соответственно) переносятся в текстовые поля `android.widget.TextView` Активности (идентификаторы `tvLastName` и `tvFirstName`). Это необходимо сделать до открепления Фрагмента от Активности. В нашем примере это делается непосредственно в методе, обрабатывающем событие нажатия на кнопку «Применить».

И последнее, Диалоговое окно, созданное с помощью переопределения метода `onCreateView()`, не закроется автоматически при нажатии на кнопки «Применить» или «Отменить» — для этой цели необходимо самостоятельно вызывать метод `dismiss()` Диалогового окна `android.app.DialogFragment`.

Теперь перейдем к варианту создания Диалогового окна с помощью переопределения метода `onCreateDialog()`. Метод выглядит следующим образом:

```
Dialog onCreateDialog (Bundle savedInstanceState);
```

Этот метод принимает объект `android.os.Bundle` (параметр `savedInstanceState`), в котором содержится последнее

сохраненное состояние Фрагмента, которое он мог сохранить в методе обратного вызова `onSaveInstanceState()`.

Метод `onCreateDialog` должен вернуть ссылку на объект Диалогового окна ([android.app.Dialog](#)), которое в нашем примере будет создаваться с помощью объекта `android.app.AlertDialog.Builder`.

Также интересно знать, что метод `onCreateDialog()` вызывается в жизненном цикле Фрагмента после вызова метода `onCreate()` и перед вызовом метода `onCreateView()`.

Класс Фрагмента нашего примера, в котором используется вариант создания Диалогового окна с помощью переопределения метода `onCreateDialog()`, называется **DialogFragmentTwo** (файл `/java/com.itstep.myapp3/DialogFragmentTwo.java`) и его содержимое представлено в Листинге 1.24.

#### Листинг 1.24. Программный код класса DialogFragmentTwo

```
package com.itstep.myapp3;

import ...

/**
 * Class DialogFragmentTwo – Пример создания
 * Диалогового окна с помощью переопределения
 * метода onCreateDialog и AlertDialog.Builder
 * -----
 */
public class DialogFragmentTwo extends DialogFragment
{
    //--- Class constants -----
    /**
     * Ключ для Bundle параметра, который содержит
     * идентификатор текстового поля TextView, из
```

```
* которого Диалоговое окно считывает
* Рост. После закрытия Диалогового окна
* по кнопке "Применить", в это текстовое поле
* будет записан Рост, который ввел пользователь
*/
public final static String DIALOGTWO_HEIGHT =
    "dialog_two_height";

/**
 * Ключ для Bundle параметра, который содержит
 * идентификатор текстового поля TextView,
 * из которого Диалоговое окно считывает Вес.
 * После закрытия Диалогового окна по кнопке
 * "Применить", в это текстовое поле будет
 * записан Вес, который ввел пользователь
*/
public final static String DIALOGTWO_WEIGHT =
    "dialog_two_weight";

//--- Class members -----
/***
 * Ссылка на текстовое поле EditText, в которое
 * пользователь будет вводить Рост
*/
private EditText edtHeight;

/***
 * Ссылка на текстовое поле EditText, в которое
 * пользователь будет вводить Вес
*/
private EditText edtWeight;

//--- Class methods -----
@Override
public Dialog onCreateDialog(Bundle
    savedInstanceState)
{
```

```
//-- Получение ссылки на объект LayoutInflater ---  
    LayoutInflater inflater = this.getActivity().  
        getLayoutInflater();  
  
//-- Создание набора виджетов для Диалогового окна  
    View view = inflater.inflate(  
        R.layout.layout_two, null, false);  
  
//-- Создание объекта android.app.AlertDialog.  
//-- Builder -----  
    AlertDialog.Builder builder =  
        new AlertDialog.Builder(this.  
            getActivity());  
  
//-- Настройка Диалогового окна с помощью  
//-- AlertDialog.Builder -----  
    builder.setTitle("Укажите рост и вес");  
    builder.setView(view);  
  
//-- Инициализация полей объекта -----  
    this.edtHeight = (EditText)  
        view.findViewById(R.id.edtHeight);  
    this.edtWeight = (EditText)  
        view.findViewById(R.id.edtWeight);  
  
//-- Назначение обработчиков событий кнопкам ----  
    builder.setNegativeButton("Отменить",  
        new DialogInterface.OnClickListener()  
    {  
        @Override  
        public void onClick(DialogInterface dialog, int which)  
        {  
        }  
    });  
  
    builder.setPositiveButton("Применить",  
        new DialogInterface.OnClickListener()
```

```
{  
    @Override  
    public void onClick(DialogInterface dialog, int which)  
    {  
        //-- Помещаем введенные пользователем Рост и Вес  
        //-- в текстовые поля Активности  
        Bundle bundleArgs = DialogFragmentTwo.  
            this getArguments();  
        if (bundleArgs.containsKey(  
            DialogFragmentTwo.  
            DIALOGTWO_HEIGHT))  
        {  
            ((TextView) DialogFragmentTwo.this.  
                getActivity().findViewById(  
                    bundleArgs.getInt(  
                        DialogFragmentTwo.  
                        DIALOGTWO_HEIGHT))).  
                setText(DialogFragmentTwo.  
                    this.edtHeight.getText());  
        }  
  
        if (bundleArgs.containsKey(  
            DialogFragmentTwo.  
            DIALOGTWO_WEIGHT))  
        {  
            ((TextView) DialogFragmentTwo.  
                this.getActivity().  
                findViewById(  
                    bundleArgs.getInt(  
                        DialogFragmentTwo.  
                        DIALOGTWO_WEIGHT))).  
                setText(DialogFragmentTwo.this.  
                    edtWeight.getText());  
        }  
    }  
});
```

```
//-- Возвращаем Диалоговое окно -----
    return builder.create();
}

@Override
public void onResume()
{
    super.onResume();

//-- Получение Bundle-переметров -----
    Bundle bundleArgs = this.getArguments();

//-- Инициализация Текстовых полей EditText
//-- Диалогового окна -----

    if (bundleArgs.containsKey(DialogFragmentTwo.
                                DIALOGTWO_HEIGHT))
    {
        this.edtHeight.setText(
            ((TextView) this.getActivity().
                findViewById(bundleArgs.getInt(
                    DialogFragmentTwo.
                    DIALOGTWO_HEIGHT))).getText());
    }

    if (bundleArgs.containsKey(DialogFragmentTwo.
                                DIALOGTWO_WEIGHT))
    {
        this.edtWeight.setText(
            ((TextView) this.getActivity().
                findViewById(bundleArgs.getInt(
                    DialogFragmentTwo.
                    DIALOGTWO_WEIGHT))).getText());
    }
}
```

В методе `onCreateDialog()` класса `DialogFragmentTwo` создается Диалоговое окно с помощью объекта `android.app.AlertDialog.Builder`. При этом, это Диалоговое окно имеет специальный пользовательский интерфейс, содержащий виджеты для ввода значений «Рост» и «Вес». Этот пользовательский интерфейс сверстан в xml файле макета `/res/layout/layout_two.xml`. Содержимое этого файла приведено в Листинге 1.25.

**Листинг 1.25.** Xml верстка макета с виджетами для содержимого Диалогового окна, которое будет создано на основе класса `DialogFragmentTwo`

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"

    android:padding="4dp"
    android:stretchColumns="*">

    <TableRow>
        <TextView
            android:text="Рост"
            android:layout_gravity="center"
            android:textSize="8pt"      />
        <EditText
            android:id="@+id/edtHeight" />
    </TableRow>

    <TableRow>
        <TextView
            android:text="Вес"
            android:layout_gravity="center"
            android:textSize="8pt"      />
        <EditText
            android:id="@+id/edtWeight" />
    </TableRow>

```

```
    android:layout_gravity="center"
    android:textSize="8pt" />

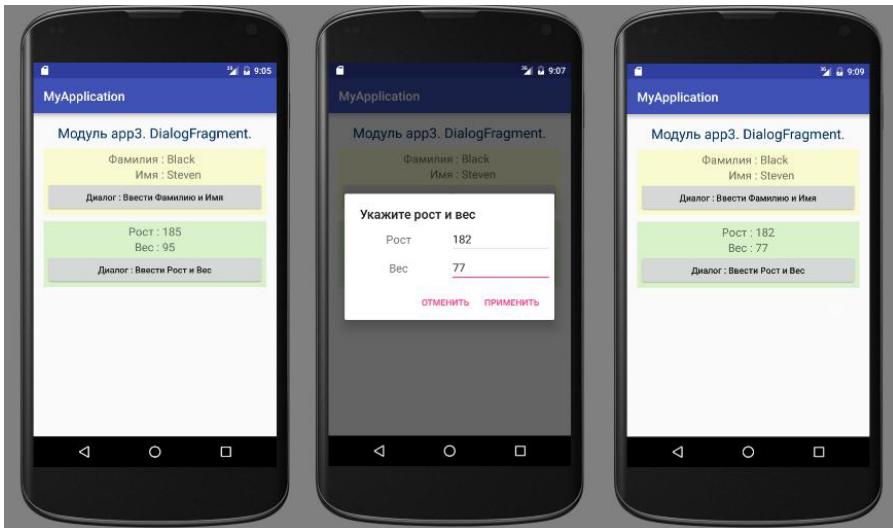
<EditText

    android:id="@+id/edtWeight" />
</TableRow>

</TableLayout>
```

Внешний вид Диалогового окна из Листингов 1.24 и 1.25 изображен на Рис. 1.9.

По аналогии с Диалоговым окном **DialogFragmentOne**, Диалоговое окно **DialogFragmentTwo** получает в качестве параметров (набор параметров в пакете **android.os.Bundle**) идентификаторы текстовых полей Активности (**tvHeight**, **tvWeight**), из которых необходимо прочитать значения для «Рост» и «Вес», чтобы затем эти значения разместить в Диалоговом окне (в полях **android.widget.EditText** с идентификаторами **edtHeight** и **edtWeight** соответственно). Передача значений идентификаторов текстовых полей осуществляется в Листинге 1.21 (второй выделенный жирным шрифтом фрагмент). Размещение значений из текстовых полей **tvHeight** и **tvWeight** в текстовые поля **edtHeight** и **edtWeight** происходит в обработчике события **onResume()** (см. Листинг 1.24). Измененные значения для значений «Рост» и «Вес» записываются обратно в текстовые поля Активности в обработчике события нажатия на позитивную кнопку «Применить» (см. Листинг 1.24). Внешний вид примера с Диалоговым окном **DialogFragmentTwo** изображен на Рис. 1.9.



**Рис. 1.9.** Внешний вид работы приложения с Диалоговым окном, создаваемым на основе класса DialogFragmentTwo (с переопределением метода onCreateDialog())

Весь исходный код примера из текущего раздела находится в модуле «app3» среди файлов с исходными кодами, которые прилагаются к данному уроку.

## 1.9. Фрагмент «Список». Класс android.app.ListFragment

Рассмотрим еще один тип Фрагментов — Список. Этот Фрагмент представлен классом [android.app.ListFragment](#) и предназначен для отображения списков. Иерархия для класса [android.app.ListFragment](#) следующая:

```
java.lang.Object
  |
  +--- android.app.Fragment
      |
      +--- android.app.ListFragment
```

Как видно из иерархии классов, класс **android.app.ListFragment** является прямым наследником от уже хорошо нам знакомого класса **android.app.Fragment**. Только в качестве дополнительной функциональности, в объекте класса **android.app.ListFragment**, инкапсулирован объект класса **android.widget.ListView**. С классом **android.widget.ListView** мы знакомились в прошлых уроках данного курса, а значит, нам будет легче познакомится с Фрагментом **android.app.ListFragment**.

Получить из Фрагмента **android.app.ListFragment** доступ к объекту **android.widget.ListView** можно с помощью метода класса **android.app.ListFragment**:

```
ListView getListView();
```

Сразу сделаем важное напоминание, что получение ссылки на объект **android.widget.ListView** можно только тогда, когда Фрагмент **android.app.ListFragment** прошел определенную часть своего Жизненного цикла (был прикреплен к Активности). Например безопасно получать ссылку на объект **android.widget.ListView** с помощью вызова метода **getListView()** можно в обработчике события Жизненного цикла Фрагмента **onResume()**.

Заполнение списка **android.app.ListFragment** можно осуществить следующими способами:

- Непосредственно заполнение списка в подчиненном объекте **android.widget.ListView**, с использованием наиболее подходящего Адаптера Данных.
- Заполнение подходящего Адаптера Данных и назначение этого Адаптера Данных объекту **android.app.**

**ListFragment**, с помощью вызова метода класса **android.app.ListFragment**:

```
void setListAdapter(ListAdapter adapter);
```

Поскольку взаимодействие с Адаптерами Данных нам уже знакомо из предыдущих уроков, давайте сразу перейдем к примеру, демонстрирующему работу Фрагмента **android.app.ListFragment**. Для примера данного раздела создан модуль «app4» в проекте с файлами исходного кода, прилагаемых к данному уроку. В качестве списка для примера приложения возьмем список электронных адресов. Элемент списка представлен классом **EmailContact** (находится в файле `/java/com.itstep.myapp4/MainActivity.java`). Хотя в данном примере будет использован Адаптер Данных [android.widget.SimpleAdapter](#) и создание специального класса **EmailContact** для элементов данных в этом случае не совсем оправдано, класс **EmailContact** создан для более удобного представления хранимых приложением данных. Код класса **EmailContact** приведен в Листинге 1.26.

**Листинг 1.26.** Класс EmailContact для элементов списка контактов электронной почты

```
/**  
 * Class EmailContact – Элемент списка контактов  
 * электронной почты  
 * -----  
 */  
class EmailContact  
{  
    //--- Class members -----
```

```
/**  
 * Фамилия  
 */  
  
public String lastName;  
  
/**  
 * Имя  
 */  
  
public String firstName;  
  
/**  
 * Адрес электронной почты  
 */  
  
public String email;  
  
//-- Class methods -----  
public EmailContact(String lNm, String fName,  
                     String email)  
{  
    this.lastName    = lNm;  
    this.firstName   = fName;  
    this.email       = email;  
}  
}
```

Как видно из Листинга 1.26, каждый элемент списка электронных адресов будет состоять из фамилии (поле **lastname**), имени (поле **firstname**) и электронного адреса абонента (поле **email**).

Макет Активности (файл `/res/layout/activity_main.xml`), в которую будет встраиваться Фрагмент `android.app.ListFragment` представлен в Листинге 1.27.

### Листинг 1.27. Макет Активности приложения для рассматриваемого примера текущего раздела

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    xmlns:tools=  "http://schemas.android.com/tools"

    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.itstep.myapp4.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textColor="#003366"
        android:textSize="10pt"
        android:layout_gravity="center_horizontal"
        android:text="Модуль app4. ListFragment." />

    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/flFragmentContainer" >

    </FrameLayout>
</LinearLayout>
```

Как видно из Листинга 1.27, Фрагмент **android.app.ListFragment** будет встраиваться в Активность программно. Для этого в Активности специально размещен контейнер **android.widget.FrameLayout** с идентификатором **flFragmentContainer** (в Листинге 1.27 выделен жирным шрифтом).

Для элемента списка электронных адресов **android.widget.ListView** создадим xml макет (файл /res/layout/email\_contact\_item.xml). Согласно верстке этого макета, каждый элемент списка будет выглядеть как изображено на Рис. 1.10.



**Рис. 1.10.** Внешний вид каждого элемента списка электронных адресов

Xml содержимое файла макета (/res/layout/email\_contact\_item.xml) с версткой внешнего вида элемента списка электронных адресов приведен в Листинге 1.28.

**Листинг 1.28.** Xml верстка макета для элементов списка электронных адресов, который будет отображаться в Фрагменте android.app.ListFragment

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"

    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="#d3dfa4"
    android:padding="4dp">

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal">
```

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="10pt"
    android:textColor="#003366"
    android:id="@+id/tvLastName" />

<Space
    android:layout_width="10dp"
    android:layout_height="wrap_content"/>

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="10pt"
    android:textColor="#003366"
    android:id="@+id/tvFirstName" />
</LinearLayout>

<Space
    android:layout_width="match_parent"
    android:layout_height="4dp"/>

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:textSize="9pt"
    android:textStyle="italic"
    android:textColor="#006633"
    android:id="@+id/tvEmail" />
</LinearLayout>
```

Как видно из Листинга 1.28, для отображения фамилии, имени и электронного адреса абонента предназначены виджет **android.widget.TextView** с идентификаторами **tvLastName**, **tvFirstName** и **tvEmail** соответственно. Эти

соответствия будут указаны при создании Адаптера Данных **android.widget.SimpleAdapter**.

Теперь перейдем к созданию Фрагмента **android.app.ListFragment** и Адаптера Данных **android.widget.SimpleAdapter** со списком электронных адресов. Создание этих объектов происходит в методе **onCreate()** класса Активности и представлено в Листинге 1.29.

### Листинг 1.29. Создание Фрагмента android.app.ListFragment

```
public class MainActivity extends AppCompatActivity
{
    //--- Class constants -----
    /**
     * Ключ для SimpleAdapter: Фамилия
     */
    private final static String ADAPTER_KEY_LASTNAME =
        "adapter_key_lastname";

    /**
     * Ключ для SimpleAdapter: Имя
     */
    private final static String ADAPTER_KEY_FIRSTNAME =
        "adapter_key_firstname";

    /**
     * Ключ для SimpleAdapter: email
     */
    private final static String ADAPTER_KEY_EMAIL =
        "adapter_key_email";

    //--- Class methods -----
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
    }
}
```

```
setContentView(R.layout.activity_main);

//--- Коллекция с данными -----
ArrayList<EmailContact> emails =
        new ArrayList<>();
emails.add(new EmailContact("White", "Steven",
    "steven@gmail.com"));
emails.add(new EmailContact("Black", "Bill",
    "bill@microsoft.com"));
emails.add(new EmailContact("Smith", "Mary",
    "mary_smith@gmail.com"));
emails.add(new EmailContact("Green", "Peter",
    "green_p@meta.com"));
emails.add(new EmailContact("Peach", "Sara",
    "sara@microsoft.com"));
emails.add(new EmailContact("Gray",
    "Serg", "grayserg@gmail.com"));
emails.add(new EmailContact("Lemon",
    "Kristina", "unknown@meta.com"));

//--- Адаптер Данных -----
ArrayList<HashMap<String, String>> items =
        new ArrayList<>();

for (int i = 0; i < emails.size(); i++)
{
    HashMap<String, String> fieldMap =
        new HashMap<>();
    EmailContact ec = emails.get(i);
    fieldMap.put(MainActivity.
        ADAPTER_KEY_LASTNAME, ec.lastName);
    fieldMap.put(MainActivity.
        ADAPTER_KEY_FIRSTNAME, ec.firstName);
    fieldMap.put(MainActivity.
        ADAPTER_KEY_EMAIL, ec.email);
    items.add(fieldMap);
}
```

```
SimpleAdapter adapter =
        new SimpleAdapter(this, items,
R.layout.email_contact_item,
        new String[]
{
        MainActivity.ADAPTER_KEY_LASTNAME,
        MainActivity.ADAPTER_KEY_FIRSTNAME,
        MainActivity.ADAPTER_KEY_EMAIL
},
        new int[]
{
        R.id.tvLastName,
        R.id.tvFirstName,
        R.id.tvEmail
});
//-- ListFragment -----
ListFragment LF = new ListFragment()
{
    @Override
    public void onResume()
    {
        super.onResume();

//-- Обработчик события клика по элементу списка
//-- в ListFragment -----
        this.getListView().setOnItemClickListener(
            new AdapterView.OnItemClickListener()
            {
                @Override
                public void onItemClick(AdapterView<?>
parent, View view,
int position, long id)
                {
                    HashMap<String, String> map =
                        (HashMap<String, String>)
parent.getAdapter()
                    .getItem(position);
                }
            }
        );
    }
}
```

```
        String str =
            "Last Name: " + map.get(
                MainActivity.ADAPTER_KEY_
                LASTNAME) + "\n" +
            "First Name: " + map.get(
                MainActivity.ADAPTER_KEY_
                FIRSTNAME) + "\n" +
            "Email: " + map.get(
                MainActivity.ADAPTER_KEY_EMAIL);

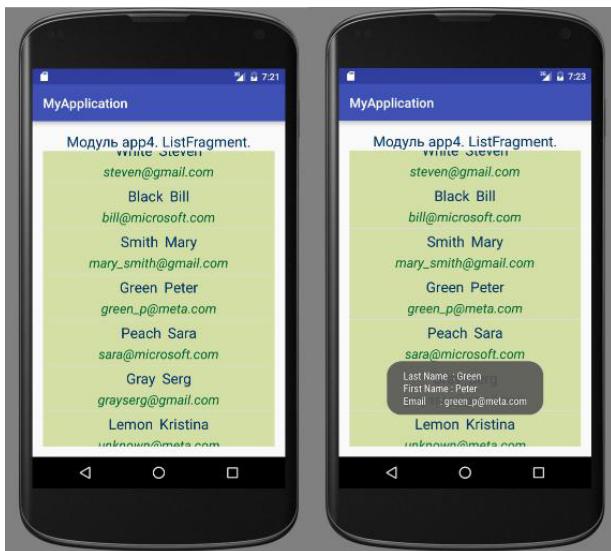
        Toast.makeText(MainActivity.
            this, str, Toast.LENGTH_SHORT).
        show();
    }
}
} ;
LF.setAdapter(adapter);

//-- FragmentManager / FragmentTransaction -----
FragmentManager FM = this.getFragmentManager();
FragmentTransaction FT = FM.beginTransaction();

FT.add(R.id.flFragmentContainer, LF);
FT.commit();
}
}
```

Внешний вид работы примера из Листинга 1.29 изображен на Рис. 1.11. В отображаемом Фрагментом списке контактов электронных адресов пользователь может нажать на элемент списка и выбранный элемент списка отобразится в объекте **Toast**. Для обработки нажатия на элемент списка используется объект, обработчик события **AdapterView.OnItemClickListener**. Назначение

обработчика события нажатия на элемент списка **android.widget.ListView** (который встроен в объект **android.app.ListFragment**) происходит в методе Жизненного цикла Фрагмента **onResume()**. Так как в методе **onCreate()** класса Активности (где происходит создание Фрагмента **android.app.ListFragment** и его прикрепление к Активности) встроенный в Фрагмент объект **android.widget.ListView** еще не существует.



**Рис. 1.11.** Внешний вид работы примера из Листинга 1.29

Весь исходный код примера текущего раздела находится в модуле "app4" проекта среди файлов исходных кодов, прилагаемых к данному уроку.

## 2. Понятие Intent. Класс android.content.Intent. Запуск Активностей других приложений

С помощью объекта [android.content.Intent](#) можно инициализировать выполнение необходимой нашему приложению функциональности с помощью [Компонента другого приложения](#). К компонентам приложения относятся:

- Активности ([android.app.Activity](#)). Что такое Активность мы уже знаем — это экран с пользовательским интерфейсом.
- Службы ([android.app.Service](#)). Служба предназначена для работы в фоновом режиме для выполнения длительных операций, связанных с работой удаленных процессов. Служба не имеет пользовательского интерфейса.
- Поставщики Контента ([android.content.ContentProvider](#)). Поставщик Контента управляет общим набором данных приложения. Данные можно хранить в файловой системе, базе данных SQLite, в Интернете или любом другом постоянном месте хранения, к которому у вашего приложения имеется доступ.
- Приемники Широковещательных Сообщений ([android.content.BroadcastReceiver](#)). Приемник Широковещательных Сообщений получает объявления распространяемые по всей системе. Многие из этих объ-

явлений рассыпает система — например объявление о том, что экран выключился, аккумулятор разряжен или был сделан фотоснимок. Объявления также могут рассыпаться приложениями, — например, чтобы сообщить другим приложениям о том, что какие-то данные были загружены на устройство и теперь готовы для использования.

Так вот, Активности, Службы и Приемники Широко-вещательных Сообщений могут запускаться с помощью объектов **android.content.Intent**. Данный раздел урока посвящен запуску Активностей других приложений с помощью объектов **android.content.Intent**.

Возникает вопрос — а зачем запускать Активности других приложений? Ответ заключается в том, что в Операционной Системе Android любое приложение может запустить компонент другого приложения. Такая возможность является особенностью ОС Android по сравнению с другими Операционными Системами. Например, если разработчик в своем приложении хочет предоставить пользователю возможность использования фотокамеры устройства для получения фотоснимков, то разработчику нет необходимости разбираться с тонкостями и сложностями взаимодействия с фотокамерой устройства, — он может воспользоваться возможностями уже имеющегося приложения и запустить Активность фотографирования из приложения для фотокамеры. По завершению процесса фотографирования, готовая фотография будет передана в приложение, которое вызвало Активность фотографирования. А для пользователя это будет выглядеть, как работа одного приложения. Это очень привлекательная для разработчиков приложений возможность.

Непосредственный запуск Активности с помощью объекта **android.content.Intent** осуществляется в два этапа:

- Создание и конфигурирование объекта **android.content.Intent**.
- Запуск Активности с помощью метода класса **android.content.Context** (который наследуют все классы Активностей):

```
void startActivity (Intent intent);
```

Иерархия класса **android.content.Intent** имеет следующий вид:

```
java.lang.Object  
|  
+--- android.content.Intent
```

Для создания объекта **android.content.Intent** необходимо воспользоваться одним из конструкторов:

- **Intent (String action)** — конструктор принимает параметр **action**, который содержит действие, которое необходимо запускающему другую Активность приложению. Значения для параметра **action** представлены в виде констант класса **android.content.Intent**, которые задают общие запрашиваемые действия присутствующие в Операционной Системе: **ACTION\_CALL** (осуществить вызов), **ACTION\_DIAL** (осуществить набор номера), **ACTION\_EDIT** (выполнить операцию редактирования данных), **ACTION\_SEARCH** (выполнить операцию поиска данных), **ACTION\_SENDTO** (выполнить отправку сообщения), **ACTION\_VIEW** (выполнить операцию просмотра данных) и так далее.

После создания объекта **android.content.Intent** с помощью этого конструктора, необходимо выполнить еще дополнительные настройки объекта **Intent** — ведь недостаточно всего лишь указать требуемое действие, необходимо еще передать объекту дополнительные **Intent** данные, над которыми необходимо выполнить действие.

- **Intent (Context packageContext, Class<?> cls)** — конструктор создает объект **Intent** с указанием контекста приложения (параметр **packageContext**), в котором находится класс Активности (параметр **cls**), объект которой необходимо запустить. В этом случае так же необходимо передать дополнительную информацию в созданный объект **Intent**.

Как уже упоминалось выше, после создания объекта **Intent** необходимо этому объекту передать дополнительную информацию, которая необходима для запуска другой Активности. В объекте **Intent** для информации, которая необходима для запуска другой Активности, предназначены следующие поля: **action**, **data**, **type**, **class**, **category**, **extras**. Давайте рассмотрим подробнее эти составляющие:

- **action** — выполняемое действие, например **ACTION\_VIEW**, **ACTION\_EDIT**, **ACTION\_MAIN** и так далее.
- **data** — данные для работы (например запись из списка контактов в виде **content://contacts/people/N**, где N — идентификатор записи из списка контактов, или в виде **tel:123-45-67**).
- **type** — определяет тип данных (MIME тип), которые содержатся в объекте **Intent**.

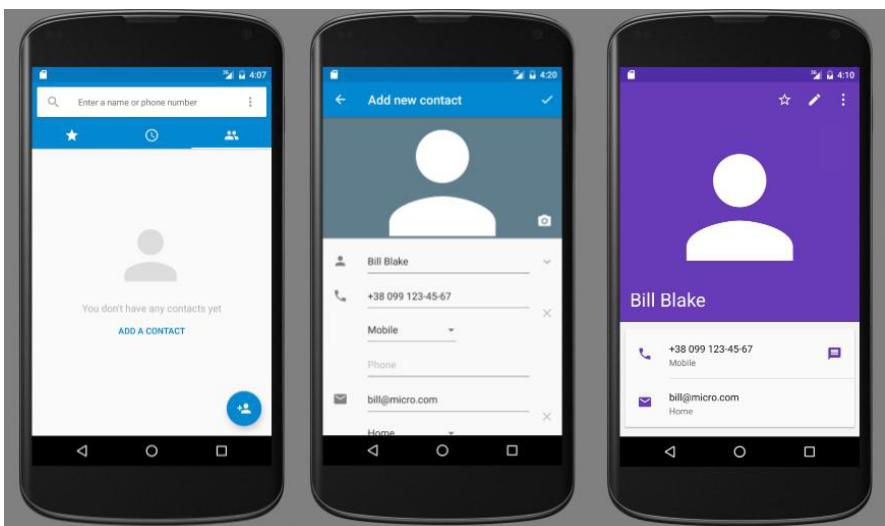
- **class** — название класса Активности, которую необходимо запустить.
- **category** — категория, содержит дополнительную информацию о действии, например **CATEGORY\_LAUNCHER**, **CATEGORY\_ALTERNATIVE** (см. описание класса [android.content.Intent](#)). Категорий в объекте **Intent** может быть несколько.
- **extras** — содержит объект **android.os.Bundle** с дополнительными данными, которые необходимы запускаемой Активности.

Для управления вышеперечисленной информацией предназначены следующие методы класса **Intent**:

- **Intent addCategory(String category)** — добавляет Категорию в список категорий объекта **Intent**.
- **Intent setAction(String action)** — устанавливает выполняемое действие для объекта **Intent**.
- **Intent setClass(Context packageContext, Class<?> cls)** — устанавливает контекст приложения и имя класса Активности.
- **Intent setType(String type)** — устанавливает MIME-тип данных объекта **Intent**.
- **Intent putExtra(String name, ТИП value)** — добавляет в объект **Bundle** данные с ключом **name**.
- **String getAction()** — возвращает выполняемое действие для объекта **Intent**.
- **Set<String> getCategories()** — возвращает коллекцию категорий объекта **Intent**.

- **Bundle getExtras()** — возвращает объект `android.os.Bundle` с дополнительными данными.
- **String getType()** — возвращает MIME-тип.
- **boolean hasCategory(String category)** — возвращает признак наличия в объекте `Intent` категорий.
- **boolean hasExtra(String name)** — возвращает признак наличия дополнительных данных (в объекте `android.os.Bundle`) в объекте `Intent`.

Рассмотрим примеры запуска Активностей с помощью объектов `Intent`. Для рассматриваемых примеров добавим пару записей в список контактов устройства. Как это можно сделать показано на Рис. 2.1. Программный код примеров данного раздела находится в модуле «app5» файлов исходного кода, прилагаемых к данному уроку.



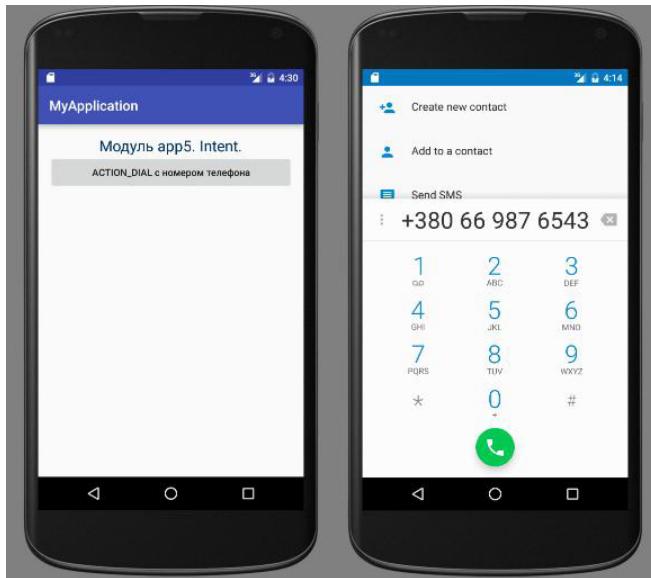
**Рис. 2.1.** Добавление контактов в список контактов устройства

Первым примером, будет запуск Активности для набора некоторого телефонного номера (действие Intent.ACTION\_DIAL с данными tel:+38 066 987-65-43). Для этого на Активности нашего приложения разместим кнопку с надписью «ACTION\_DIAL с номером телефона» и идентификатором **btnDialOne**. Обработчиком события нажатия на эту и другие кнопки рассматриваемых примеров сделаем метод **btnClick()**. При нажатии на эту кнопку должна появиться Активность для набора телефонного номера с уже введенным номером телефона «+38 066 987-65-43». Программный код запуска такой Активности с помощью объекта **android.content.Intent** приведен в Листинге 2.1.

**Листинг 2.1.** Запуск с помощью объекта android.content.Intent  
Активности набора телефонного номера  
(действие Intent.ACTION\_DIAL)

```
public void btnClick(View v)
{
    switch (v.getId())
    {
        //--- Запуск Активности ACTION_DIAL с телефонным
        //--- номером -----
        case R.id.btnDialOne:
        {
            Intent oneIntent = new Intent();
            oneIntent.setAction(Intent.ACTION_DIAL);
            oneIntent.setData(Uri.
                parse("tel:+38 066 987-65-43"));
            this.startActivity(oneIntent);
        }
        break;
        ...
    }
}
```

Внешний вид работы примера из Листинга 2.1 изображен на Рис. 2.2.



**Рис. 2.2.** Внешний вид работы примера из Листинга 2.1

**Примечание:** Для осуществления звонков нашему приложению необходимо добавить в файл манифеста приложения *manifests/AndroidManifest.xml* специальное разрешение:

```
<uses-permission android:name="android.permission.  
CALL_PHONE" />
```

Следующим шагом запустим другую Активность, которая покажет нам список контактов. Для этого на Активность нашего приложения добавим кнопку с надписью «ACTION\_VIEW списка контактов» и идентификатором **btnViewOne**. При клике на эту кнопку появится

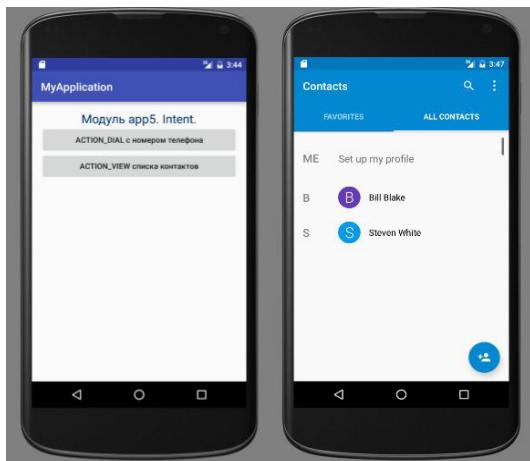
Активность, в которой будет отображен список контактов нашего устройства. Для запускаемой Активности действие будет Intent.ACTION\_VIEW, а данными будет строка «content://contacts/people». Программный код запуска такой Активности с помощью объекта android.content.Intent приведен в Листинге 2.2.

### Листинг 2.2. Запуск с помощью объекта android.content.Intent Активности просмотра всего списка контактов

```
public void btnClick(View v)
{
    switch (v.getId())
    {
        ...
        //-- Запуск Активности ACTION_VIEW со списком
        //-- контактов -----
        case R.id.btnViewOne:
        {
            Intent oneIntent = new Intent();
            oneIntent.setAction(Intent.ACTION_VIEW);
            oneIntent.setData(Uri.
                parse("content://contacts/people"));
            this.startActivity(oneIntent);
        }
        break;
    }
}
```

**Примечание:** Для формирования строк доступа к контенту (например «content://contacts/people») или для получения информации о контенте прочитайте самостоятельно информацию «Поставщики контента» [по ссылке](#).

Внешний вид работы примера из Листинга 2.2 изображен на Рис. 2.3.



**Рис. 2.3.** Внешний вид работы примера из Листинга 2.2

Аналогично запустим Активность для просмотра (действие **Intent.ACTION\_VIEW**) конкретного контакта из списка контактов устройства. Для этого в данные объекта **android.content.Intent** необходимо разместить строку следующего формата: **content://contacts/people/N**, где **N** — идентификатор контакта из списка контактов.

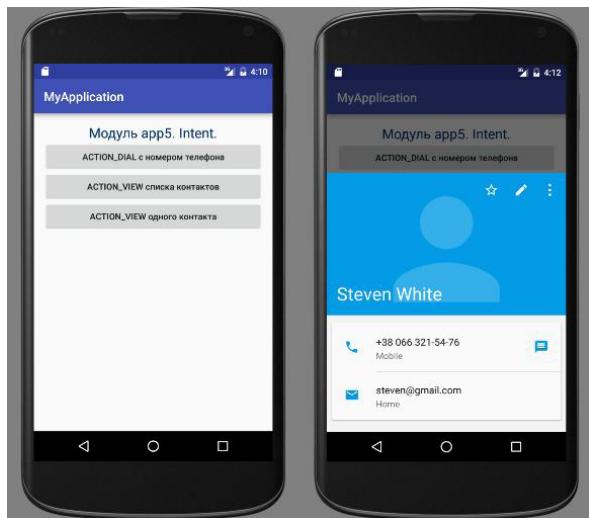
В нашем примере добавим на Активность нашего приложения кнопку с надписью «ACTION\_VIEW одного контакта» и идентификатором **btnViewTwo**. Программный код запуска такой Активности с помощью объекта **android.content.Intent** приведен в Листинге 2.3.

### **Листинг 2.3.** Запуск с помощью объекта android.content.Intent Активности просмотра конкретного контакта

```
public void btnClick(View v)
{
    switch (v.getId())
    {
```

```
...
    //--- Запуск Активности ACTION_VIEW с контактом из
    //--- списка контактов -----
    case R.id.btnViewTwo:
    {
        Intent oneIntent = new Intent();
        oneIntent.setAction(Intent.ACTION_VIEW);
        oneIntent.setData(Uri.parse("content://
            contacts/people/2"));
        this.startActivity(oneIntent);
    }
    break;
}
}
```

Внешний вид работы примера из Листинга 2.3 изображен на Рис. 2.4. Как видно из Рис. 2.4, кроме просмотра контакта, пользователю предоставляется возможность внести изменения в контакт (пиктограмма «Карандаш»).



**Рис. 2.4.** Внешний вид работы примера из Листинга 2.3

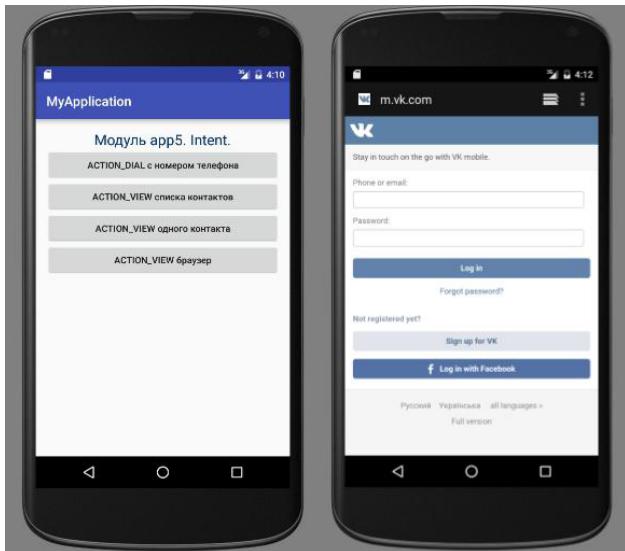
Следующий пример — открыть ссылку в браузере. В нашем приложении на Активность добавим кнопку с надписью «ACTION\_VIEW браузер» и идентификатором **btnViewThree**. Для запуска Активности «Браузер» необходимо создать объект **android.content.Intent** с действием **Intent.ACTION\_VIEW** и данными, в которых будет содержаться адрес ссылки, например «<http://www.vk.com>». Пример запуска Активности «Браузер» с указанной ссылкой приведен в Листинге 2.4.

**Листинг 2.4.** Запуск Активности,  
чтобы просмотреть ссылку в браузере

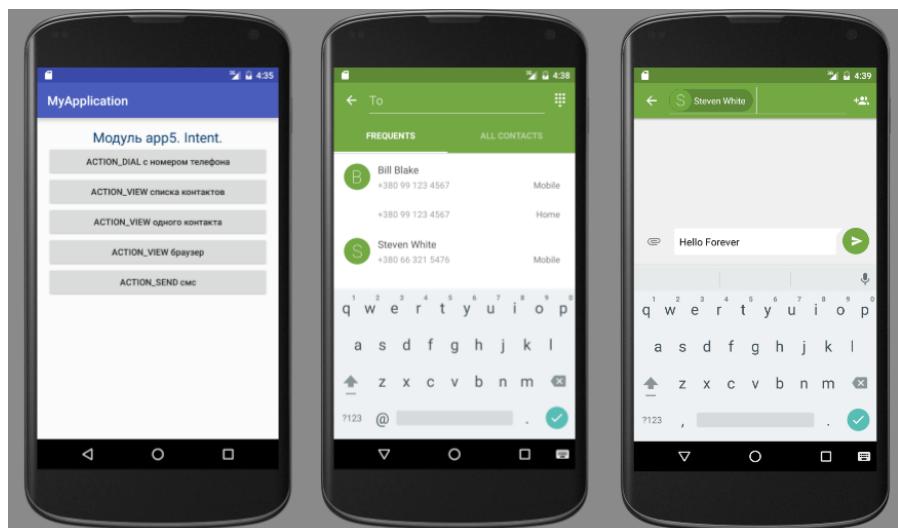
```
public void btnClick(View v)
{
    switch (v.getId())
    {
        ...
        //-- Запуск Активности ACTION_VIEW Открыть ссылку
        //-- в браузере -----
        case R.id.btnViewThree:
        {
            Intent oneIntent = new Intent();
            oneIntent.setAction(Intent.ACTION_VIEW);
            oneIntent.setData(Uri.
                parse("http://www.vk.com"));
            startActivity(oneIntent);
        }
        break;
    }
}
```

Внешний вид работы примера из Листинга 2.4 изображен на Рис. 2.5.

## Урок №4



**Рис. 2.5.** Внешний вид работы примера из Листинга 2.4



**Рис. 2.6.** Внешний вид работы примера из Листинга 2.5

Следующий пример — отправить смс-сообщение. В Активность нашего приложения добавим кнопку с надписью «ACTION\_SEND смс» и идентификатором **btnSendOne**. Текст смс-сообщения «Hello Forever» установим в запускаемую Активность программно. Пример приведен в Листинге 2.5. Внешний вид работы примера из Листинга 2.5 изображен на Рис. 2.6.

### **Листинг 2.5.** Запуск Активности отправки смс сообщения

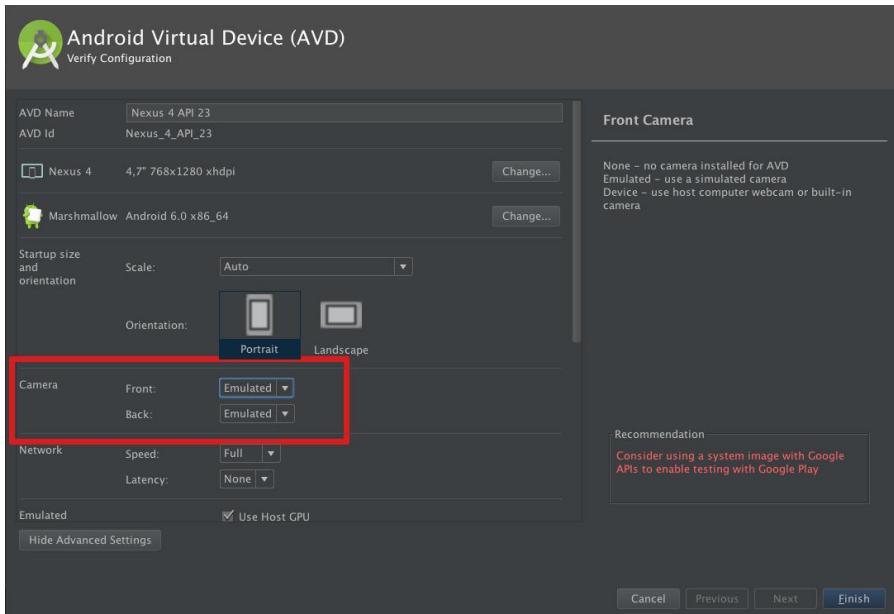
```
public void btnClick(View v)
{
    switch (v.getId())
    {
        ...
        //-- Запуск Активности отправки смс-сообщения --
        case R.id.btnSendOne:
        {
            Intent smsIntent = new Intent(Intent.
                ACTION_SEND);
            smsIntent.setType("text/plain");
            smsIntent.putExtra(Intent.EXTRA_TEXT,
                "Hello Forever");
            startActivity(smsIntent);
        }
        break;
    }
}
```

Как видно из Рис. 2.6, программный код Листинга 2.5 приводит к запуску Активности отправки смс-сообщения, где вначале пользователю предоставляется возможность выбрать получателя сообщения из списка контактов, а затем появляется окно с текстовым полем для ввода

текста сообщения, в котором уже находится строка сообщения «Hello Forever», которую пример из Листинга 2.5 разместил с помощью кода, который выделен жирным шрифтом (см. Листинг 2.5).

Для каждого действия (`Intent.ACTION_XXXX`) есть набор дополнительных параметров, которые программист может дополнительно разместить в объекте `android.content.Intent`, для достижения более удобной функциональности запускаемой Активности. Информацию о настройках и дополнительных параметрах для объекта `android.content.Intent` для каждого действия `Intent.ACTION_XXXX` необходимо смотреть в API документации для класса [android.content.Intent](#). Конечно, в наших примерах демонстрируются лишь некоторые возможности для запуска Активностей с действиями `Intent.ACTION_VIEW`, `Intent.ACTION_SEND`, `Intent.ACTION_DIAL`. Поэтому для более полного изучения интересующих вас возможностей, вам необходимо более детальное изучение API документации по данному вопросу.

И последний пример, который мы рассмотрим в этом разделе, будет пример запуска Активности и получения результата выполнения этой Активности нашим приложением, с целью дальнейшей обработки этого полученного результата. Для этого воспользуемся Активностью получения фотоснимков с камеры устройства. Чтобы пример успешно отработал, необходимо включить для виртуального устройства эмуляцию Камеры. Для этого нужно запустить утилиту «AVD Manager» (Менеджер



**Рис. 2.7.** Включение эмуляции Камеры виртуального устройства для примера запуска Активности получения фотоснимков

Виртуальных Устройств), затем выбрать виртуальное устройство, которое вы используете для запуска своих приложений и примеров данного урока, и в дополнительных настройках Виртуального Устройства включить эмуляцию Камеры (см. Рис. 2.7). Если к вашему компьютеру подключена Web-камера, то можно указать в настройках, что для эмуляции Камеры устройства можно использовать Web-камеру вашего компьютера. Если же Web-камеры у вас нет, то необходимо выбрать программную эмуляцию, как показано на Рис. 2.7.

Перед тем, как рассматривать программный код примера, ознакомимся с механизмом запуска Активностей,

с целью дальнейшего получения в наше приложение результата от них. Для запуска подобных Активностей используется метод класса **android.app.Activity**:

```
void startActivityForResult (Intent intent,  
    int requestCode);
```

Этот метод запускает с помощью объекта **android.content.Intent** (параметр **intent**) другую Активность, от которой необходимо получить результат в виде каких-либо данных. Второй параметр **requestCode** – это некоторое числовое значение, которое будет идентифицировать получаемый результат в методе **onActivityResult()**. Значение для параметра **requestCode** программист придумывает самостоятельно и размещает (рекомендуется) в виде константы, например в классе Активности приложения.

Когда пользователь с помощью запущенной другой Активности получит необходимые ему данные и закроет эту другую Активность, в нашем приложении будет вызван метод обратного вызова (класс **android.app.Activity**):

```
void onActivityResult (int requestCode,  
    int resultCode, Intent data);
```

В этом методе наше приложение и получит результат исполнения другой Активности. Метод **onActivityResult()** принимает следующие параметры:

- **requestCode** — числовое значение, которое идентифицирует полученные данные. Это значение было передано в метод **startActivityForResult()** в качестве одноименного параметра **requestCode**.

- **resultCode** — код завершения другой Активности. Например значение **Activity.RESULT\_CANCELLED** означает, что пользователь отказался получать данные от другой Активности и закрыл ее. Кодом успешного завершения будет значение **Activity.RESULT\_OK**.
- **data** — полученные от другой Активности данные. Формат данных зависит от вида запускаемой Активности и описывается в API документации для разработчиков Android приложений.

Для Активности, получающей фотоснимок с Камеры устройства, полученными данными будет объект **android.graphics.Bitmap**, который будет содержать изображение сделанного фотоснимка.

Теперь настало время рассматривать пример. Сразу же создадим целочисленную константу, которая будет идентифицировать получаемый от другой Активности результат. Объявление константы показано в Листинге 2.6.

**Листинг 2.6.** Объявление в классе Активности приложения константы, которая будет идентифицировать получаемый от другой Активности результат

```
public class MainActivity extends AppCompatActivity
{
    //--- Class constants -----
    /**
     * Целочисленная константа, идентифицирующая
     * получаемый результат от Активности, делающей
     * фотоснимки
     */
    private final static int REQUEST_CAMERA = 987;
    ...
}
```

Далее, на Активность нашего примера добавим кнопку с текстом «ACTION\_IMAGE\_CAPTURE» и идентификатором **btnImageCapture**, а также виджет **android.widget.ImageView** (идентификатор **ivImage**), в котором будем отображать полученные от другой Активности фотоснимки. Xml разметка этих виджетов приведена в Листинге 2.7.

**Листинг 2.7.** Xml разметка виджетов, добавляемых на Активность рассматриваемого примера, для получения фотоснимка от другой Активности

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    ...
    tools:context="com.itstep.myapp5.MainActivity">

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textAllCaps="false"
        android:id="@+id	btnImageCapture"
        android:text="ACTION_IMAGE_CAPTURE"
        android:onClick="btnClick"
    />

    <ImageView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/ivImage"
    />

</LinearLayout>
```

Программный код, обрабатывающий событие нажатия на кнопку с надписью «ACTION\_IMAGE\_CAPTURE»,

который запускает Активность получения фотоснимков приведен в Листинге 2.8.

### Листинг 2.8. Запуск Активности для получения фотоснимков

```
public void btnClick(View v)
{
    switch (v.getId())
    {
        ...
        //--- Запуск Активности получения фотоснимков ---
        case R.id.btnImageCapture:
        {
            Intent oneIntent = new
                Intent(MediaStore.
                    ACTION_IMAGE_CAPTURE);
            startActivityForResult(oneIntent,
                MainActivity.REQUEST_CAMERA);
        }
        break;
    }
}
```

Напомним, что просто запустить Активность в данном случае недостаточно. Нам необходимо получить результат в виде изображения полученной фотографии. Как уже описывалось выше, для получения результата от других Активностей, необходимо в классе Активности нашего приложения переопределить метод обратного **onActivityResult()**, который будет вызван автоматически при закрытии другой Активности. Код метода обратного вызова **onActivityResult()** нашего рассматриваемого примера приведен в Листинге 2.9.

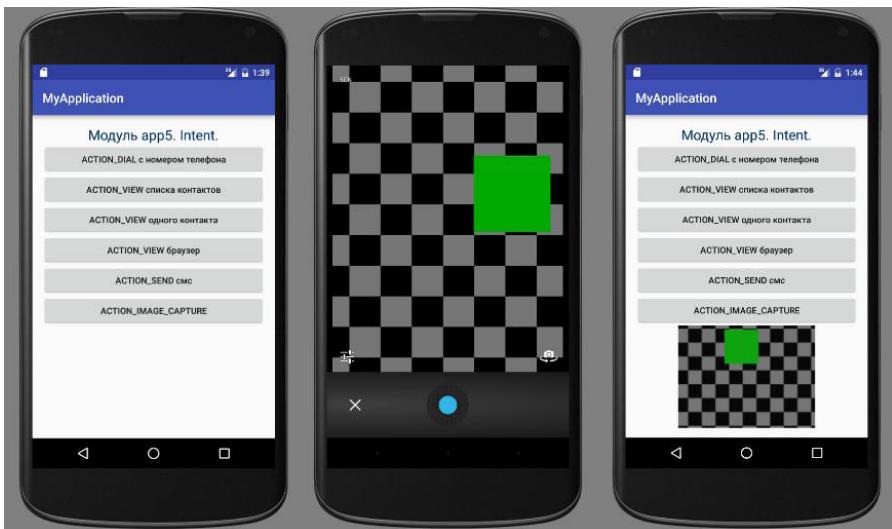
**Листинг 2.9.** Метод обратного вызова `onActivityResult()`,  
который получает фотоснимок от Активности  
получения фотоснимков

```
public class MainActivity extends AppCompatActivity
{
    ...
    @Override
    protected void onActivityResult(int requestCode,
                                    int resultCode, Intent data)
    {
        if (requestCode == MainActivity.REQUEST_CAMERA)
        {
            if (resultCode == Activity.RESULT_OK)
            {
                Bitmap img = (Bitmap) data.getExtras() .
                    get("data");
                ImageView ivImage = (ImageView)
                    findViewById(R.id.ivImage);
                ivImage.setImageBitmap(img);
            }
        }
    }
}
```

Метод `onActivityResult()` может быть использован для получения данных от разных Активностей. Поэтому (см. Листинг 2.9) первым делом метод `onActivityResult()` определяет, от какой Активности получены данные. В нашем примере для этой цели используется константа `MainActivity.REQUEST_CAMERA`. Затем необходимо проверить, что пользователь подтвердил успешное закрытие другой Активности. Для этого параметр `resultCode` должен быть равен значению `Activity.RESULT_OK` (а не `Activity.RESULT_CANCEL`). И если условие совпадает, то

можно извлекать полученные данные. В нашем примере (см. Листинг 2.9) из данных извлекается объект `android.graphics.Bitmap` и размещается в виджете `android.widget.ImageView`.

Внешний вид работы примера из Листингов 2.7, 2.8, 2.9 изображен на Рис. 2.8.



**Рис. 2.8.** Внешний вид работы примера из Листингов 2.7, 2.8, 2.9

На Рис. 2.8 изображена следующая последовательность действий: пользователь нажимает на кнопку «ACTION\_IMAGE\_CAPTURE», появляется Активность получения фотоснимков. В нашем примере эмулируется работа Камеры устройства в виде перемещающегося квадрата, зеленого или красного цвета, по шахматке. Далее, пользователь жмет на голубую круглую кнопку «Сделать снимок» и возвращается в Активность нашего приложения, где в виджете `android.widget.ImageView` с идентификатором

**ivImage** отображается сделанный fotosнимок. При необходимости, можно дополнить данный пример сохранением полученного fotosнимка на устройство внешнего носителя. Вам предлагается сделать это самостоятельно.

Программный код примеров данного раздела находится в модуле «app5» файлов с исходными кодами, которые прилагаются к данному уроку.

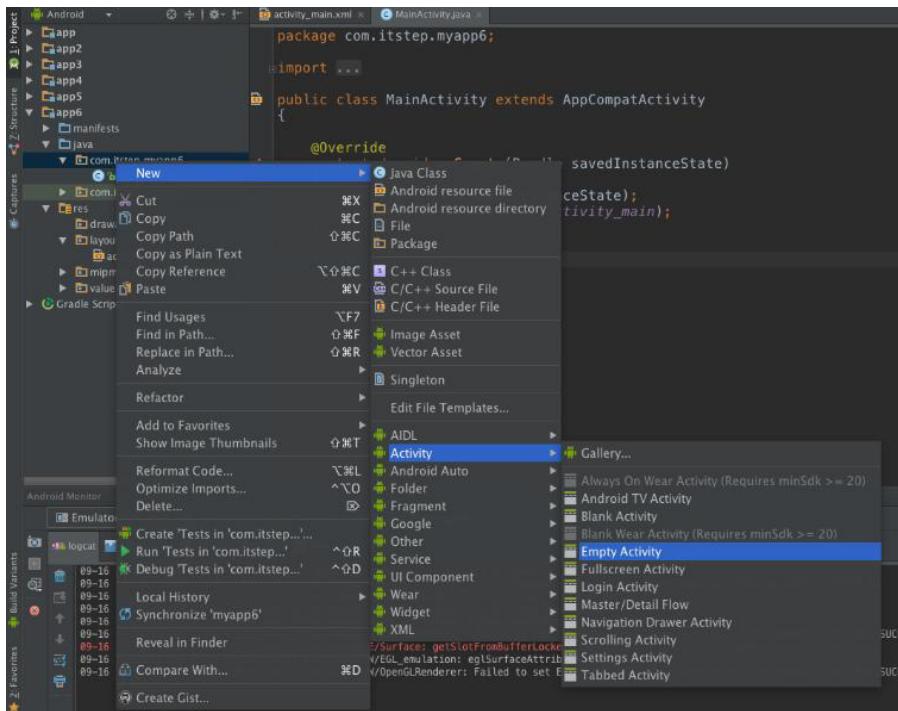
## 3. Создание второй Активности в приложении и ее запуск

В предыдущем разделе мы рассматривали механизмы запуска Активностей из других приложений. В этом разделе будет рассмотрен механизм создания дополнительных Активностей в нашем приложении и запуск таких Активностей. Ведь очень многие Android-приложения состоят из нескольких Активностей, в чем вы смогли убедиться из предыдущего раздела данного урока. Да и возможность самостоятельно создавать Активности, которые могут быть запущены другими приложениями, также выглядит привлекательной.

Для того чтобы в нашем приложении появилась вторая Активность, необходимо сделать 3 шага:

- Создать в проекте (а точнее — в модуле) Java-класс, производный от класса `android.app.Activity` (или от любого из его производных классов).
- Создать в проекте xml файл макета внешнего вида Активности.
- Добавить информацию о второй Активности в Манифест.

К счастью, интегрированная среда разработки Android Studio позволяет упростить процесс создания второй (третьей, четвертой и так далее) Активности в нашем проекте. Для этого необходимо в Android Studio кликнуть

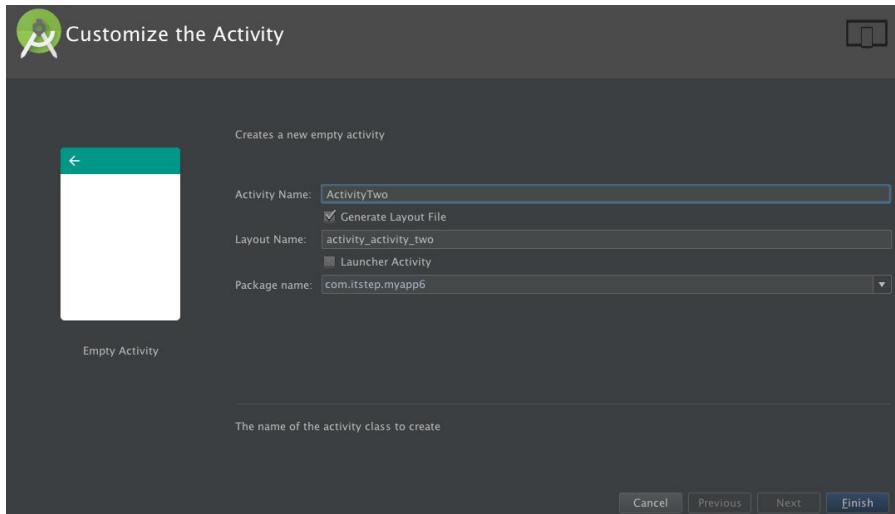


**Рис. 3.1. Добавление в модуль "app6" второй Активности на основе Empty Activity**

правой кнопкой мыши по вкладке нашего проекта и в появившемся меню выбрать пункты «New» / «Activity» / «Активность». В нашем примере будет создана Пустая Активность (Empty Activity), как изображено на Рис. 3.1.

Итак, выберем, как изображено на Рис. 3.1, пункты меню «New» / «Activity» / «Empty Activity». Появится диалоговое окно (см. Рис. 3.2), в котором необходимо будет указать имя класса второй Активности (опция «Activity Name») и название файла (опция «Layout Name») с xml макетом для нее (от файла xml макета можно отказаться в этом же диалоговом окне с помощью опции «Generate

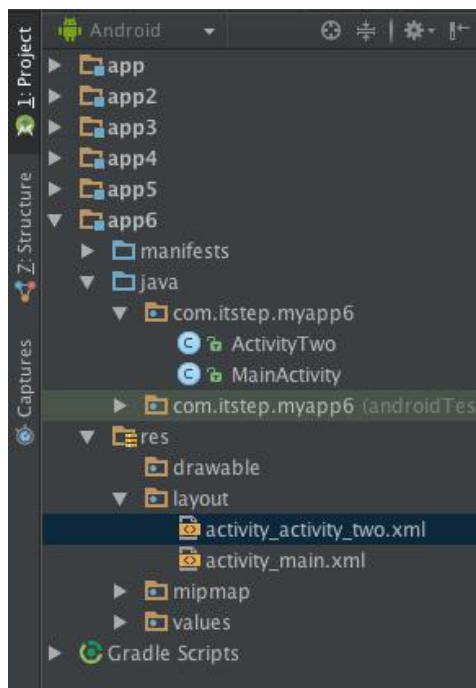
**Layout File»). Нашей создаваемой второй Активности дадим название класса **ActivityTwo**, а файлу xml макета — **activity\_activity\_two** (как изображено на Рис. 3.2)**



**Рис. 3.2.** Второй шаг создания второй Активности в приложении. Назначение названий для класса Активности и xml файла макета

Далее жмем на кнопку «Finish» и во вкладке «Project», для нашего модуля «app6», видим (см. Рис. 3.3) появление файлов /java/com.itstep.myapp6/ActivityTwo.java и /res/layout/activity\_activity\_two.xml.

Созданные файлы /java/com.itstep.myapp6/ActivityTwo.java и /res/layout/activity\_activity\_two.xml ничем не отличаются от аналогичных файлов для первой Активности, которые создает Android Studio при создании модуля. Для наглядности, в Листинге 3.1 приведен исходный код класса **ActivityTwo** (файл /java/com.itstep.myapp6/ActivityTwo.java).



**Рис. 3.3.** Добавленные в модуль "app6" проекта файлов /java/com.itstep.myapp6/ActivityTwo.java и /res/layout/activity\_activity\_two.xml для второй Активности

**Листинг 3.1.** Сгенерированный Android Studio класс ActivityTwo для второй Активности приложения

```
package com.itstep.myapp6;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class ActivityTwo extends AppCompatActivity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_activity_two);
    }
}
```

А вот в Манифесте приложения для модуля «аррб» произошли интересные изменения. Код Манифеста (файл /manifests/AndroidManifests.xml) приведен в Листинге 3.2.

### Листинг 3.2. Манифест приложения с информацией о второй Активности приложения

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.itstep.myapp6"
    xmlns:android=
        "http://schemas.android.com/apk/res/android">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.
                    action.MAIN"/>

                <category android:name="android.intent.
                    category.LAUNCHER"/>
            </intent-filter>
        </activity>

        <activity android:name=".ActivityTwo">
        </activity>
    </application>
</manifest>
```

В Листинге 3.2 жирным шрифтом выделен фрагмент кода, в котором указана информация, что в приложении есть еще одна Активность. Позже мы вернемся к этому фрагменту кода, чтобы внести настройки `<intent-filter>` для второй Активности.

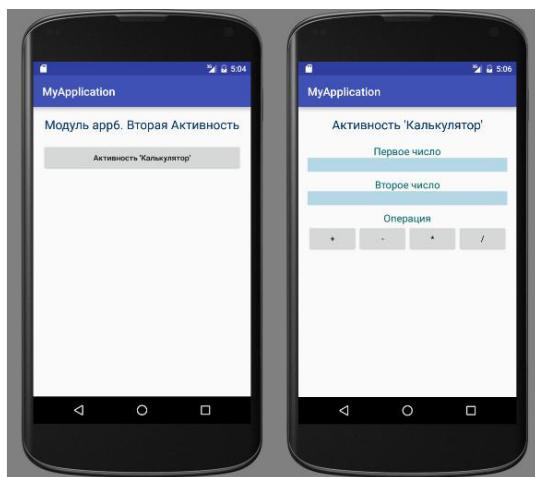
Сверстаем xml макет внешнего вида для второй Активности приложения. Чтобы сконцентрироваться на механизме запуска второй Активности и не отвлекаться на функциональность, которую будет выполнять эта вторая Активность, пусть это будет Активность «Калькулятор». Простейший калькулятор с двумя текстовыми полями — для ввода чисел (идентификаторы `edtOne` и `edtTwo` соответственно) и четырьмя кнопками — для простейших математических операций (идентификаторы `btnPlus`, `btnMinus`, `btnMult`, `btnDiv`). Код верстки xml макета для второй Активности, ввиду понятности, в Листингах данного урока не приводится. Активность «Калькулятор» будет возвращать результат вычисления в вызывающую Активность, с помощью применения методов `startActivityForResult()` и `onActivityResult()`.

Для начала просто запустим вторую Активность, без получения какого-либо результата. Для этого на главную Активность нашего приложения добавлена кнопка с надписью «Активность "Калькулятор"» и идентификатором `btnCalc`. Метод, который будет обрабатывать событие нажатия на кнопку, называется `btnClick()`. При нажатии на кнопку `btnCalc` и произойдет запуск Активности «Калькулятор». Программный код запуска второй Активности приведен в Листинге 3.3.

#### Листинг 3.3. Запуск второй Активности нашего приложения

```
public class MainActivity extends AppCompatActivity
{
    ...
    public void btnActivityClick(View v)
    {
        switch (v.getId())
        {
            case R.id.btnCalc:
                {
                    Intent intent = new Intent(this,
                        ActivityTwo.class);
                    startActivityForResult(intent);
                }
                break;
        }
    }
}
```

Внешний вид работы примера из Листинга 3.3 изображен на Рис. 3.4.



**Рис. 3.4.** Внешний вид работы примера из Листинга 3.3

Как видно из Листинга 3.3, для запуска второй Активности нашего приложения «Калькулятор» использовался объект **android.content.Intent**, который был создан с помощью конструктора, принимающий в качестве второго параметра класс (тип **Class<T>**) Активности, которую нужно запустить. Такой способ запуска Активностей мы еще не использовали. Напомним, что при создании в Манифесте записи для второй Активности (класс **ActivityTwo**) было указано имя класса (см. Листинг 3.2, код выделенный жирным шрифтом). И если при создании объекта **android.content.Intent** мы укажем тот же класс, то система, просмотрев Манифест, обнаружит соответствие и запустит соответствующую Активность. Если убрать выделенную жирным шрифтом в Листинге 3.2 запись из Манифеста, то при попытке запуска второй Активности произойдет исключительная ситуация:

```
android.content.ActivityNotFoundException:  
    Unable to find explicit activity class  
    {com.itstep.myapp6/com.itstep.myapp6.ActivityTwo};
```

Такой запуск второй Активности с использованием имени класса называется «Явным вызовом» Активности и действует в пределах одного приложения. Существует также и «Неявный вызов» Активности. При неявном вызове не указывается имя класса Активности, а указываются параметры: действие (**action**), данные (**data**), категорию (**category**) и тип (**type**).

Кстати, все примеры запуска Активностей из предыдущего раздела данного урока относятся к неявному вызову Активностей.

Так вот, чтобы была возможность осуществлять неявный вызов Активности, необходимо в Манифесте приложения для Активности указать параметры (те же самые **action**, **data**, **category**), которые может принимать Активность через объект **android.content.Intent**. Это делается в Манифесте приложения с помощью элементов **<intent-filter>** для Активности. Элементы **<intent-filter>** так и называют фильтрами. Каждый фильтр **<intent-filter>** указывает тип объектов **android.content.Intent**, которые принимает Активность на основании действия (**action**), данных (**data**) и категории (**category**), заданных в объекте **android.content.Intent**. Операционная Система Android передаст неявный объект **android.content.Intent** приложению, только если он (объект **android.content.Intent**) может пройти через один из фильтров **<intent-filter>**, указанных в Манифесте приложения.

Добавим, что если Операционная Система Android найдет среди всех приложений только одну Активность, удовлетворяющую параметрам **action**, **data**, **category**, то будет запущена именно эта найденная Активность. В случае, если Операционная Система Android найдет несколько Активностей, чьи фильтры проходит объект **android.content.Intent**, то пользователю будет представлен список для выбора, в котором он может сам выбрать, какое приложение ему использовать. Если же Операционная Система не найдет ни одну из Активностей, чьим фильтром удовлетворяет объект **android.content.Intent**, то будет сгенерирована исключительная ситуация, и наше приложение завершит свою работу аварийно. Поэтому, перед попыткой неявного запуска

Активности, необходимо осуществить проверку, что существует Активность, чьим фильтрам удовлетворяет наш объект **android.content.Intent**. Это можно сделать, например с помощью метода класса **android.content.Intent**:

```
ComponentName resolveActivity (PackageManager pm);
```

Этот метод вернет ссылку на объект **android.content.ComponentName**, который содержит Активность, которая будет запущена, или null, если не найдена Активность, фильтрам которой удовлетворяет наш объект **android.content.Intent**.

Метод **resolveActivity()** принимает ссылку на объект **android.content.pm.PackageManager**, который предназначен для получения всевозможной информации о приложениях, установленных на устройстве. Получить объект **android.content.pm.PackageManager** можно с помощью метода класса **android.content.Context**:

```
PackageManager getPackageManager ();
```

Пример неявного запуска Активности с проверкой, что найдена Активность, чьи фильтры прошел наш объект **android.content.Intent**, представлен в Листинге 3.4.

**Листинг 3.4.** Пример проверки, что для объекта **android.content.Intent** найдена Активность

```
Intent sendIntent = new Intent();  
sendIntent.setAction(Intent.ACTION_SEND);  
sendIntent.putExtra(Intent.EXTRA_TEXT,  
        "This is Sparta!");
```

```
sendIntent.setType("text/plain");
//-- Проверяем, что объект android.content.Intent
//-- прошел фильтр хотя бы одной Активности ----

if (sendIntent.resolveActivity(getApplicationContext())
    != null)
{
    startActivity(sendIntent);
}
```

Вернемся к Манифесту приложения и к элементам `<intent-filter>`. У элемента `<intent-filter>` есть три дочерних элемента: `<action>`, `<data>`, `<category>`. Рассмотрим, какую информацию нужно в них размещать:

- `<action>` — объявляет принимаемое действие, заданное в объекте `android.content.Intent`. Значение для действия задается в атрибуте `name` этого элемента, в виде строки, содержащей название действия, а не название константы класса (т. е. не значение «`Intent.ACTION_SEND`», а значение «`android.intent.action.SEND`»).
- `<data>` — объявляет тип принимаемых данных. Значение передается в атрибуте `imeType` элемента.
- `<category>` — объявляет принимаемую категорию, заданную в объекте `android.content.Intent`, в атрибуте `name`. Значение должно быть текстовой строкой категории, а не константой класса (т. е. не значение «`Intent.CATEGORY_DEFAULT`», а значение «`Android.intent.category.DEFAULT`»).

Пример создания такого фильтра приведен в Листинге 3.5.

### Листинг 3.5. Пример фильтра <intent-filter> с описанием элементов action, data, category

```
<activity android:name="SomeActivity">
    <intent-filter>

        <action android:name="android.intent.
            action.SEND"/>
        <category android:name="android.intent.
            category.DEFAULT"/>
        <data android:mimeType="text/plain"/>

    </intent-filter>
</activity>
```

Теперь, с учетом полученных знаний, создадим в Манифесте фильтр для нашей второй Активности «Калькулятор». На данный момент неважно, какое действие (**action**), категорию (**category**) и тип данных (**data**) мы укажем в фильтре. Важно, чтобы Операционная Система Android нашла нашу Активность «Калькулятор» при неявном запуске. Назначим **action**, **category**, **data** для Активности «Калькулятор» нашего примера те же значения, которые указаны в Листинге 3.5. С этими значениями наша Активность «Калькулятор» будет совпадать с параметрами Активности отправки смс-сообщений. Посмотрим, что получится. Внесем изменения в Манифест приложения (см. Листинг 3.6) и добавим на главную Активность нашего примера еще одну кнопку с надписью «"Калькулятор" неявный вызов» и идентификатором **btnCalcTwo**. Пример неявного запуска Активности «Калькулятор» приведен в Листинге 3.7 (обработчик события нажатия на кнопку **btnCalcTwo**).

**Листинг 3.6.** Манифест приложения с настроенным фильтром для Активности «Калькулятор». Настройки фильтра совпадают с Активностью «Отправка смс-сообщений» другого приложения

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.itstep.myapp6"
    xmlns:android=
        "http://schemas.android.com/apk/res/android">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name=
                    "android.intent.action.MAIN"/>

                <category android:name=
                    "android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>

        <activity android:name=".ActivityTwo">
            <intent-filter>
                <action android:name=
                    "android.intent.action.SEND"/>
                <category android:name=
                    "android.intent.category.DEFAULT"/>
                <data android:mimeType="text/plain"/>
            </intent-filter>
        </activity>

    </application>
</manifest>
```

В Листинге 3.6 жирным шрифтом выделены несенные в Манифест приложения изменения.

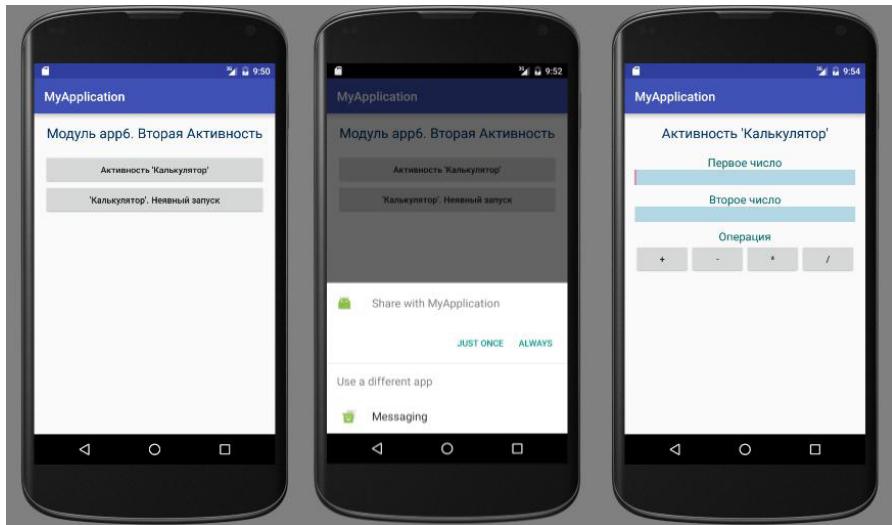
### **Листинг 3.7. Неявный запуск Активности «Калькулятор» нашего приложения или «Отправка смс-сообщения» другого приложения**

```
public void btnActivityClick(View v)
{
    switch (v.getId())
    {
        ..
        //--- Неявный запуск Активности "Калькулятор" или
        //--- "Отправка смс" -----
        case R.id.btnCalcTwo:
        {
            Intent intent = new Intent();
            intent.setAction(Intent.ACTION_SEND);
            intent.addCategory(Intent.CATEGORY_DEFAULT);
            intent.setType("text/plain");
            startActivity(intent);
        }
        break;
    }
}
```

Внешний вид работы примера из Листингов 3.6, 3.7 изображен на Рис. 3.5.

Как видно из Рис. 3.5, Операционная Система Android находит две Активности, чьи фильтры прошел наш объект **android.content.Intent** из Листинга 3.7. В этом случае Операционная Система Android показывает список для выбора наиболее подходящей для пользователя Активности. На Рис. 3.5 видно, как и ожидалось, в список попали Активность «Калькулятор» нашего приложения и Активность «Messaging» приложения отправки смс-сообщений.

### 3. Создание второй Активности в приложении и ее запуск



**Рис. 3.5.** Внешний вид работы примера из Листингов 3.6, 3.7

Аналогично, если другое приложение создаст объект `android.content.Intent` с такими же параметрами, как в Листинге 3.7, то пользователь в появившемся списке выбора увидит Активность «`Messaging`» приложения отправки смс-сообщений и нашу Активность «`Калькулятор`».

Теперь рассмотрим пример, в котором Активность будет возвращать результат в вызывающую Активность. В нашем случае возвращать результат будет Активность «`Калькулятор`». Для того, чтобы Активность вернула результат, ей нужно выполнить следующую последовательность действий:

- Создать другой объект `android.content.Intent`.
- Записать в этот объект возвращаемые данные, с помощью вызова метода класса `android.content.Intent`:

```
Intent putExtra(String name, Тип value);
```

- Для установки кода завершения работы Активности (**resultCode**) вызвать метод класса **android.app.Activity**:

```
void setResult (int resultCode, Intent intent);
```

- Для закрытия Активности вызвать метод класса **android.app.Activity**:

```
void finish();
```

Ну, и в вызывающей Активности должен быть переопределён метод **onActivityResult()**, в который в качестве параметра и будет доставлен объект **Intent** из метода **setResult()**, с результирующими данными выполнения Активности.

Для примера получения результирующих данных от Активности «Калькулятор» в нашем приложении добавим кнопку с надписью «"Калькулятор"». Получение результата» и идентификатором **btnCalcThree**. Так же на главную Активность нашего приложения добавим текстовое поле **android.widget.TextView** с идентификатором **tvCalcResult**, в которое и будет размещаться результат, полученный от Активности «Калькулятор». Так же, в Манифесте приложения, создадим еще один элемент **<intent-filter>** для того, чтобы задать уникальное значение действия (**action**) для Активности «Калькулятор» (класс **ActivityTwo**). Это делается с целью продемонстрировать, что существует и такой способ задания фильтра, который обеспечит запуск только нашей Активности, ведь если значение действия (**action**)

уникально, то совпадений с другими Активностями не произойдет. В этом случае даже не обязательно указывать значения категории (**category**) и типа данных (**data**).

Изменения, которые внесем в Манифест приложения, отображены в Листинге 3.8 (выделены жирным шрифтом).

**Листинг 3.8.** Альтернативный фильтр `<intent-filter>` для Активности «Калькулятор» (класс `ActivityTwo`)

```
<activity android:name=".ActivityTwo">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name=
            "android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
    <intent-filter>
        <action android:name=
            "com.itstep.myapp6.ActivityTwo" />
        <category android:name=
            "android.intent.category.DEFAULT"/>
    </intent-filter>
</activity>
```

Программный код обработчика события нажатия на кнопку «Калькулятор». Получение результата» (метод `btnActivityClick()`) приведен в Листинге 3.9.

**Листинг 3.9.** Запуск Активности «Калькулятор» для получения результата

```
public void btnActivityClick(View v)
{
    switch (v.getId())
    {
        ..
    }
}
```

```
//-- Запуск Активности "Калькулятор" для
//-- получения результата -----
case R.id.btnCalcThree:
{
    Intent intent = new Intent();
    intent.setAction("com.itstep.myapp6.
        ActivityTwo");
    intent.addCategory(Intent.
        CATEGORY_DEFAULT);
    this.startActivityForResult(intent,
        MainActivity.CALC_RESULT);
}
break;

}
}
```

Как видно в Листинге 3.9, в качестве значения действия (**action**) для объекта **android.content.Intent** используется строка «**com.itstep.myapp6.ActivityTwo**», которая была указана в качестве значения для элемента **<action>** фильтра **<intent-filter>** для Активности **ActivityTwo** в Манифесте приложения (см. Листинг 3.8).

Также (см. Листинг 3.9), в классе главной Активности приложения (класс **MainActivity**) объявлена целочисленная константа (**MainActivity.CALC\_RESULT**), которая будет идентифицировать получение результата от Активности «Калькулятор». Объявление этой константы приведено в Листинге 3.10.

**Листинг 3.10.** Объявление константы

MainActivity.CALC\_RESULT, которая будет идентифицировать получение результата от Активности «Калькулятор»

```
public class MainActivity extends AppCompatActivity
{
    //--- Class constants -----
    /**
     * Константа, которая будет идентифицировать
     * получение результата
     * от Активности "Калькулятор"
     */
    private final static int CALC_RESULT = 787;
    ..
}
```

Теперь перейдем к рассмотрению самого интересного кода нашего примера — кода класса **ActivityTwo** Активности «Калькулятор». Код этого класса приведен в Листинге 3.11.

**Листинг 3.11.** Код класса ActivityTwo Активности «Калькулятор»

```
public class ActivityTwo extends AppCompatActivity
{
    //--- Class constants -----
    /**
     * Ключ для размещения в объект
     * android.content.Intent результата вычисления
     */
    public final static String KEY_CALC_RESULT =
        "key_calc_result";
    //--- Class members -----
    /**
     * Текстовое поле для ввода первого числа
     */
```

```
private EditText edtOne;

/**
 * Текстовое поле для ввода второго числа
 */
private EditText edtTwo;

//--- Class methods -----
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_activity_two);

//--- Инициализация полей объекта -----
    this.edtOne = (EditText)
        this.findViewById(R.id.edtOne);
    this.edtTwo = (EditText)
        this.findViewById(R.id.edtTwo);
}

public void btnCalcClick(View v)
{
//--- Получаем данные из текстовых полей
//--- android.widget.EditText ---
    double one;
    double two;
    double result = 0;

//--- Получаем первое число -----
    try
    {
        one = Double.parseDouble(this.edtOne.
            getText().toString());
    }
    catch (Exception e)
```

```
{  
    one    = 0;  
}  
  
//--- Получаем второе число -----  
try  
{  
    two = Double.parseDouble(this.edtTwo.  
        getText().toString());  
}  
catch (Exception e)  
{  
    two = 0;  
}  
  
//--- Выполняем математическую операцию -----  
switch (v.getId())  
{  
    case R.id.btnAdd:  
        result = one + two;  
        break;  
  
    case R.id.btnMinus:  
        result = one - two;  
        break;  
  
    case R.id.btnMult:  
        result = one * two;  
        break;  
  
    case R.id.btnDiv:  
        result = (two != 0)?(one / two):0;  
        break;  
}  
  
//--- Формируем данные для возврата в другую Активность  
Intent resultIntent = new Intent();
```

```
        resultIntent.putExtra(ActivityTwo.  
            KEY_CALC_RESULT, String.valueOf(result));  
    this.setResult(Activity.RESULT_OK,  
        resultIntent);  
    this.finish();  
  
}  
}
```

Кнопкам «+» (сложение, идентификатор **btnPlus**), «-» (вычитание, **btnMinus**), «\*» (умножение, **btnMult**), «/» (деление, **btnDiv**) назначен обработчик события нажатия метод **btnCalcClick()**. Код (см. Листинг 3.11) этого метода вполне стандартный — получение данных из текстовых полей **android.widget.EditText** и выполнение математической операции — должен быть нам понятен. Создание объекта **android.content.Intent** и размещение в нем результирующих данных исполнения Активности и передача этих данных в вызывающую Активность в Листинге 3.11 выделен жирным шрифтом.

Также обратите внимание, что для размещаемых в объекте **android.content.Intent** данных, с помощью метода **putExtra()**, используется ключ **ActivityTwo.KEY\_CALC\_RESULT**, который объявлен в качестве строковой константы в классе **ActivityTwo**, чтобы вызывающая Активность могла воспользоваться этим ключом для извлечения результирующих данных.

Программный код класса **MainActivity** (главная Активность) получения результирующих данных от Активности «Калькулятор» расположен в методе **onActivityResult()** и приведен в Листинге 3.12.

**Листинг 3.12.** Код метода onActivityResult() получения результата от Активности «Калькулятор» и размещения результата в текстовом поле android.widget.TextView с идентификатором tvCalcResult

```
@Override
public void onActivityResult(int requestCode,
                             int resultCode, Intent data)
{
    //-- Если ответ пришел от Активности "Калькулятор"
    if (requestCode == MainActivity.CALC_RESULT)
    {
        if (resultCode == Activity.RESULT_OK)
        {

            //-- Получим результат -----
            String result = data.getStringExtra(
                ActivityTwo.KEY_CALC_RESULT);

            //-- Отобразим результат в Тосте -----
            Toast.makeText(this, "Результат: " +
                           result, Toast.LENGTH_SHORT).show();

            //-- Отобразим результат в Текстовом Поле -----
            TextView tvCalcResult = (TextView)
                this.findViewById(R.id.tvCalcResult);
            tvCalcResult.setText(result);
        }
    }
}
```

Поскольку мы рассматриваем пример создания Активности (в нашем примере это **ActivityTwo**), которая возвращает результат в вызывающую Активность, то будет интересно сразу же рассмотреть пример передачи значений из одной Активности в другую (в нашем случае передача данных будет из вызывающей Активности **MainActivity**)

в Активность **ActivityTwo**). Для передачи дополнительных данных так же используется метод **putExtra()** объекта **android.content.Intent**.

Пусть в нашем примере главная Активность **MainActivity** будет передавать в Активность «Калькулятор» два случайных вещественных числа, которые будут отображаться в Активности «Калькулятор» в текстовых полях для ввода значений.

В классе **ActivityTwo** объявим строковые константы для ключей, которые будут использоваться при передачи данных в методе **putExtra()** (который будет вызываться в классе **MainActivity**) и при извлечении этих данных в классе **ActivityTwo** при помощи вызова метода **getStringExtra()**. Объявление ключей для методов **putExtra()** и **getStringExtra()** приведено в Листинге 3.13.

**Листинг 3.13.** Объявление ключей для передачи данных в вызываемую Активность «Калькулятор»

```
public class ActivityTwo extends AppCompatActivity
{
    //--- Class constants -----
    /**
     * Ключ для размещения в объект
     * android.content.Intent результата вычисления
     */
    public final static String KEY_CALC_RESULT =
        "key_calc_result";

    /**
     * Ключ, для извлечения первого числа,
     * передаваемого в Активность
     */
}
```

```

public final static String KEY_CALC_ONE =
    "key_calc_one";

/**
 * Ключ, для извлечения второго числа,
 * передаваемого в Активность
 */
public final static String KEY_CALC_TWO =
    "key_calc_two";

```

Извлечение Активностью «Калькулятор» передаваемых для нее двух вещественных чисел, осуществляется в методе **onCreate()** и приведено в Листинге 3.14.

#### **Листинг 3.14. Извлечение Активностью «Калькулятор» передаваемых для нее данных**

```

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

    //--- Инициализация полей объекта -----
    this.edtOne = (EditText)
        this.findViewById(R.id.edtOne);
    this.edtTwo = (EditText)
        this.findViewById(R.id.edtTwo);

    //--- Получаем переданные в Активность данные
    //--- через объект Intent -----
    Intent intent = this.getIntent();
    String strOne = intent.
        getStringExtra(ActivityTwo.KEY_CALC_ONE);
    if (strOne != null)
    {

```

```
        this.edtOne.setText(strOne);  
    }  
  
    String strTwo = intent.  
        getStringExtra(ActivityTwo.KEY_CALC_TWO);  
    if (strTwo != null)  
    {  
        this.edtTwo.setText(strTwo);  
    }  
}
```

Обратите внимание, что вызываемая Активность получает ссылку на объект `android.content.Intent`, который был использован для ее запуска, с помощью вызова метода класса `android.app.Activity`:

```
Intent getIntent();
```

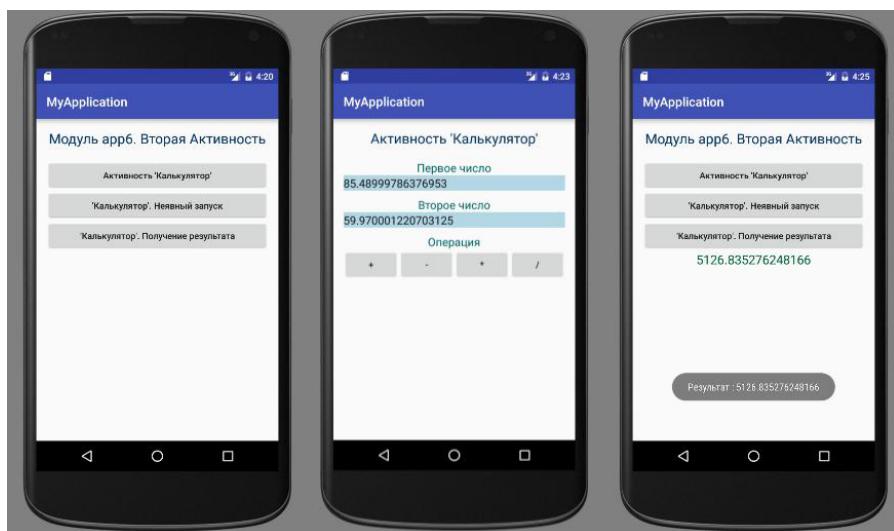


Рис. 3.6. Внешний вид работы примера из Листингов 3.8–3.15

Вызывающая Активность **MainActivity** формирует данные для передачи их в Активность «Калькулятор» непосредственно перед запуском Активности «Калькулятор» (в нашем примере в методе обработчике события нажатия **btnActivityClick()**). Этот код приведен в Листинге 3.15.

**Листинг 3.15.** Передача данных в вызываемую Активность «Калькулятор»

```
public void btnActivityClick(View v)
{
    switch (v.getId())
    {
        ..
        //-- Запуск Активности "Калькулятор"
        //--- для получения результата -----
        case R.id.btnCalcThree:
        {
            Intent intent = new Intent();
            intent.setAction("com.itstep.myapp6.
                ActivityTwo");
            intent.addCategory(Intent.CATEGORY_DEFAULT);
            double one = ((int)(Math.random() * 10000))
                / 100.0f;
            double two = ((int)(Math.random() * 10000))
                / 100.0f;
            intent.putExtra(ActivityTwo.KEY_CALC_ONE,
                String.valueOf(one));
            intent.putExtra(ActivityTwo.KEY_CALC_TWO,
                String.valueOf(two));

            this.startActivityForResult(intent,
                MainActivity.CALC_RESULT);
        }
        break;
    }
}
```

Внешний вид работы примера (Листинги 3.8–3.15), демонстрирующего обмен данными между Активностями, изображен на Рис. 3.6.

Полный исходный код примера данного раздела находится в модуле «appb» проекта с файлами исходного кода, которые прилагаются к данному уроку.

# 4. Взаимодействие Android приложений с СУБД SQLite

Настало время познакомиться с набором инструментов, которые позволяют создавать Android-приложения, которые взаимодействуют с СУБД (СУБД — *Система Управления Базами Данных*). СУБД, которая установлена по умолчанию на Android-устройства, является СУБД SQLite. Эта СУБД очень во многом похожа на СУБД MySQL, но только немножко «легче» — отсюда и слово «Lite» в названии этой СУБД. Поэтому, если вы знакомы с СУБД MySQL, то работа с СУБД SQLite не доставит вам сложностей.

Минимальное знакомство с СУБД SQLite начнем с ознакомлением с базовыми типами данных. Наиболее часто используемые типы данных СУБД SQLite:

- **INTEGER** — целочисленное значение, хранящееся в 1, 2, 3, 4, 6 или 8 байтах, в зависимости от величины самого значения.
- **REAL** — числовое значение с плавающей точкой. Хранится в формате 8-байтного числа с плавающей точкой.
- **TEXT** — значение строки текста. Хранится с использованием кодировки базы данных (UTF-8, UTF-16BE или UTF-16LE).
- **BLOB** — значение бинарных данных, хранящихся точно в том же виде, в каком были введены.
- **DECIMAL(N, M)** — числовое значение с фиксированной точкой.

Далее, для создания столбца первичного ключа, указывается спецификация: **NOT NULL PRIMARY KEY**. Ну, а если необходимо сделать столбец первичного ключа «счетчиком» (автоматическое увеличение значения в столбце), то используется спецификация **AUTOINCREMENT**.

Для ключевых слов SQL в СУБД SQLite регистр символов значения не имеет.

Пример SQL команды создания таблицы Продуктов (Products) для СУБД SQLite приведен в Листинге 4.1.

#### **Листинг 4.1.** Пример SQL запроса создания таблицы Продуктов (Products) для СУБД SQLite

```
CREATE TABLE Products
(
    id integer not null primary key autoincrement,
        -- Идентификатор
    name text,    -- Название продукта
    price real,   -- Стоимость продукта
    weight int    -- Вес продукта
);
```

Если вы знакомы в языком SQL, то код из Листинга 4.1 не должен вызывать у вас вопросов. Примеры остальных команд SQL для СУБД SQLite (**INSERT**, **UPDATE**, **DELETE**, **DROP**, **ALTER**) в уроке приводиться не будут, так как ничем не отличаются от команд SQL других СУБД.

Первым шагом, которое Android приложение должно сделать для работы с СУБД SQLite, является использование класса **android.database.sqlite.SQLiteOpenHelper**. Как именно использовать этот класс рассказывается в следующем разделе.

## 4.1. Класс android.database.sqlite.SQLiteOpenHelper

Класс [android.database.sqlite.SQLiteOpenHelper](#) является специальным (и к тому же абстрактным) классом, предназначенным для управления создания Баз Данных и их таблиц, а также для управления версиями Баз Данных.

Иерархия классов для класса [android.database.sqlite.SQLiteOpenHelper](#) достаточно простая:

```
java.lang.Object
 |
 +-- android.database.sqlite.SQLiteOpenHelper
```

Разработчику в своем приложении необходимо создать производный класс от класса [android.database.sqlite.SQLiteOpenHelper](#) и переопределить в нем следующие методы:

- **void onCreate (SQLiteDatabase db)** — метод вызывается один (!!!) раз, только при создании Базы Данных. В этом методе необходимо создать таблицы Базы Данных и, если необходимо, записать начальные значения в эти таблицы. Метод принимает ссылку на объект [android.database.sqlite.SQLiteDatabase](#) (параметр **db**), с помощью которого осуществляются посылки запросов в Базу Данных. Этот класс будет рассмотрен в следующем разделе.
- **void onUpgrade (SQLiteDatabase db, int oldVersion, int newVersion)** — метод вызывается, когда необходимо выполнить изменения в структуре Базы Данных (например добавить или удалить таблицы и т.д.). Этот метод вызывается только тогда, когда произошла смена номера версии Базы Данных. О том, как это происхо-

дит, будет рассказано ниже. Метод принимает ссылку на объект `android.database.sqlite.SQLiteDatabase` (параметр `db`), с помощью которого посылаются SQL запросы в Базу Данных, а так же номер старой версии Базы Данных (параметр `oldVersion`) и номер текущей версии (параметр `newVersion`), до которой нашему приложению необходимо «обновиться».

- `void onOpen (SQLiteDatabase db)` — метод не обязательный для переопределения. Вызывается, когда произошло подключение приложения к Базе Данных и после методов `onCreate()`, `onUpgrade()`. Метод принимает ссылку на объект `android.database.sqlite.SQLiteDatabase` (параметр `db`), с помощью которого посылаются SQL запросы в Базу Данных.

Также рекомендуется создать в производном от класса `android.database.sqlite.SQLiteOpenHelper` классе конструктор, который бы вызывал конструктор родительского класса:

```
SQLiteOpenHelper (Context context, String name,  
    SQLiteDatabase.CursorFactory factory, int version);
```

В который передаются следующие параметры:

- `Context context` — ссылка на объект `android.content.Context` текущего приложения.
- `String name` — название Базы Данных, к которой приложению необходимо выполнить подключение. Если такой Базы Данных на устройстве нет, то она будет создана и после этого вызовется метод обратного вызова `onCreate(SQLiteDatabase db)`, в котором

наше приложение должно создать структуру только что созданной Базы Данных.

- **SQLiteDatabase.CursorFactory** — ссылка на объект, реализующий интерфейс [android.database.sqlite.SQLiteDatabase.CursorFactory](#), который будет использоваться для создания Курсоров. Что такое «Курсор» будет рассказано в этом уроке ниже.
- **int version** — версия Базы Данных. Если версия Базы Данных, указанная в этом параметре, отличается от версии Базы Данных на устройстве, то вызовется метод обратного вызова **void onUpgrade (SQLiteDatabase db, int oldVersion, int newVersion)**, если значение версии больше, или вызовется метод **void onDowngrade (SQLiteDatabase db, int oldVersion, int newVersion)**, если значение версии меньше.

Так вот, рекомендуется создать в нашем производном от класса [android.database.sqlite.SQLiteOpenHelper](#) классе конструктор, который бы вызывал только что рассмотренный конструктор, с целью правильного контроля названия Базы Данных и ее версии. То есть, эти значения наш производный класс самостоятельно передаст в объект [android.database.sqlite.SQLiteOpenHelper](#), лишив пользователя нашего класса возможности допустить ошибку в названии Базы Данных или в номере ее версии, при создании объекта нашего производного класса. Как это делается приведено в Листинге 4.2.

Пример создания для нашего приложения производного класса ([MySQLiteOpenHelper](#)) от класса [android.database.sqlite.SQLiteOpenHelper](#) приведен в Листинге 4.2.

**Листинг 4.2.** Создание производного класса от класса android.database.sqlite.SQLiteOpenHelper, с переопределением методов обратного вызова для обработки событий управления Базой Данных

```
class MySQLiteOpenHelper extends SQLiteOpenHelper
{
    //-- Class constants -----
    /**
     * Tag для Log.d
     */
    private final static String TAG = "===== MySQLite";
    /**
     * Название Базы Данных
     */
    private final static String dbName = "MyDbOne";
    /**
     * Версия Базы Данных
     */
    private final static int dbVersion = 1;
    //-- Class methods -----
    public MySQLiteOpenHelper(Context context)
    {
        super(context, MySQLiteOpenHelper.dbName,
              null, MySQLiteOpenHelper.dbVersion);
    }
    /**
     * Метод обратного вызова. Вызывается, когда
     * База Данных создается впервые на устройстве.
    
```

```
* В этом методе необходимо создать таблицы
* Базы Данных и, при необходимости, заполнить
* их начальными значениями.
* @param db Объект android.database.sqlite.
* SQLiteDatabase, с помощью которого можно
* отсылать SQL запросы к Базе Данных.
*/
@Override
public void onCreate (SQLiteDatabase db)
{
    Log.d(TAG, "onCreate: " + db.getPath());

    /*
     * Создание таблиц
     * -----
     */
//-- Создание таблицы Products -----
    String query = "CREATE TABLE Products(" +
        "id integer not null primary key autoincrement, " +
        "name text, " + "price real, " +
        "weight integer)";

    db.execSQL(query);
}

/**
 * Метод обратного вызова вызывается, когда
 * База Данных нуждается в обновлении, в связи
 * с изменением номера версии Базы Данных на
 * более новую.
 * В этом методе необходимо выполнить действия
 * по изменению структуры Базы Данных:
 * удалить таблицы, добавить таблицы и т.д.
 * @param db Объект android.database.sqlite.
* SQLiteDatabase, с помощью которого можно
* отсылать SQL запросы к Базе Данных.
```

```
* @param oldVersion Номер предыдущей версии
* Базы Данных.
* @param newVersion Номер текущей версии
* Базы Данных.
*/
@Override
public void onUpgrade (SQLiteDatabase db,
                      int oldVersion, int newVersion)
{
    Log.d(TAG, "onUpgrade: " + db.getPath() +
           "; oldVersion: " + oldVersion +
           "; newVersion: " + newVersion);
}

/**
 * Метод обратного вызова, не обязательный для
 * переопределения.
 * Вызывается, когда произошло подключение
 * приложения к Базе Данных и
 * после методов onCreate(), onUpgrade().
 * @param db Объект android.database.sqlite.
 * SQLiteDatabase,
 * с помощью которого можно отсыпать SQL запросы
 * к Базе Данных.
*/
@Override
public void onOpen(SQLiteDatabase db)
{
    Log.d(TAG, "onOpen: " + db.getPath());
}

/**
 * Метод обратного вызова, не обязательный для
 * переопределения. Вызывается, когда База Данных
 * нуждается в обновлении в связи с изменением
 * номера версии Базы Данных на более старую.
```

```
* В этом методе необходимо выполнить действия по
* изменению структуры Базы Данных:
* удалить таблицы, добавить таблицы и т.д.
* @param db Объект android.database.sqlite.
* SQLiteDatabase, с помощью которого можно
* отсылать SQL запросы к Базе Данных.
* @param oldVersion Номер предыдущей версии
* Базы Данных.
* @param newVersion Номер текущей версии
* Базы Данных.
*/
@Override
public void onDowngrade (SQLiteDatabase db,
                        int oldVersion, int newVersion)
{
    Log.d(TAG, "onDowngrade: " + db.getPath() +
           "; oldVersion: " + oldVersion +
           "; newVersion: " + newVersion);
}
}
```

В Листинге 4.2 в созданном классе **MySQLOpenHelper** переопределены все методы обратного вызова с целью более подробного ознакомления с обработкой событий управления Базы Данных. В методе **onCreate()** осуществляется создание таблицы Продуктов (**Products**), как было показано в Листинге 4.1. В остальных методах пока только осуществляется вывод сообщений в Лог приложения, с помощью вызова метода **Log.d()**. Не торопитесь запускать пример из Листинга 4.2, так как обработчик события **onCreate()** вызовется только один раз при создании Базы Данных (и больше вызываться не будет).

Как уже сообщалось выше, для отправки SQL запросов в Базу Данных предназначен объект **android.database.sqlite**.

**SQLiteDatabase.** В методы обратного вызова (**onCreate()**, **onOpen()**, **onUpgrade()**) для класса производного от класса **android.database.sqlite.SQLiteOpenHelper** ссылка на объект класса **android.database.sqlite.SQLiteDatabase** передается в качестве параметра (см. Листинг 4.2). Во всех остальных случаях, необходимо получать ссылку на объект **android.database.sqlite.SQLiteDatabase** с помощью методов объекта **android.database.sqlite.SQLiteOpenHelper**:

- **SQLiteDatabase getReadableDatabase()** — создает или открывает Базу Данных, название и версия, которой содержатся в объекте **android.database.sqlite.SQLiteOpenHelper**. База Данных открывается в режиме «Только чтение».
- **SQLiteDatabase getWritableDatabase()** — создает или открывает Базу Данных, название и версия которой содержатся в объекте **android.database.sqlite.SQLiteOpenHelper**. База Данных открывается в режиме «Чтение и запись».

Пример получения ссылки на объект **android.database.sqlite.SQLiteDatabase** с помощью объекта класса **MySQLiteOpenHelper** (см. Листинг 4.2) приведен в Листинге 4.3.

#### **Листинг 4.3.** Пример получения объекта **android.database.sqlite.SQLiteDatabase**

```
MySQLiteOpenHelper dbHelper =
        new MySQLiteOpenHelper(this);
SQLiteDatabase db = dbHelper.
        getWritableDatabase();
```

Давайте теперь познакомимся с классом `android.database.sqlite.SQLiteDatabase`, с помощью которого приложение сможет полноценно взаимодействовать с Базой Данных, а так же со вспомогательными классами `android.content.ContentValues`, `android.database.sqlite.SQLiteDatabase` и затем вновь вернемся к примеру из Листинга 4.2.

## 4.2. Класс `android.content.ContentValues`

Класс `android.content.ContentValues` является вспомогательным классом для класса `android.database.sqlite.SQLiteDatabase`, так как используется при операциях добавления и обновления строк таблиц Базы Данных.

Объект `android.content.ContentValues` содержит в себе данные одной строки таблицы и представляет из себя коллекцию пар «Ключ-Значение». «Ключом» является название столбца таблицы, а «Значением» является значение этого столбца для строки, которая хранится в объекте `android.content.ContentValues`.

Иерархия классов для класса `android.content.ContentValues` имеет следующий вид:

```
java.lang.Object
 |
 +-- android.content.ContentValues
```

Объект `android.content.ContentValues` можно создать с помощью конструктора по умолчанию. Методы класса `android.content.ContentValues`:

- `void clear()` — очищает коллекцию `android.content.ContentValues`.

- `boolean containsKey(String key)` — проверяет, находится ли в коллекции значение с ключом `key`.
- `Тип getAsТип(String key)` — возвращает значение соответствующего типа (int, String, boolean, long и т.д.) из коллекции по ключу `key`.
- `void put(String key, Тип value)` — помещает значение (параметр `value`) соответствующего `типа` (int, String, boolean, long и т.д.) в коллекцию `android.content.ContentValues` с ключом `key`. Другими словами, этот метод добавляет значение `value` для столбца таблицы с именем `key`.
- `void putNull(String key)` — помещает значение `null` для столбца таблицы с именем `key`.
- `int size()` — возвращает количество значений в коллекции.
- `void remove(String key)` — удаляет значение из коллекции для ключа `key`.
- `Set<Entry<String, Object>> valueSet()` — возвращает набор пар «Ключ-Значение» из коллекции `android.content.ContentValues`.
- `Set<String> keySet()` — возвращает набор с названием ключей (названием столбцов таблицы) из коллекции `android.content.ContentValues`.

Пример создания строки для таблицы Продуктов (`Products`) с использованием объекта `android.content.ContentValues` с целью добавления этой строки в таблицу `Products` Базы Данных приведен в Листинге 4.4.

**Листинг 4.4.** Пример создания с помощью объекта android.content.ContentValues строки для добавления ее в таблицу Базы Данных с помощью объекта android.content.SQLiteDatabase

```
ContentValues row = new ContentValues();
row.put("name", "Snickers");
row.put("price", 12.50);
row.put("weight", 45);
```

### 4.3. Класс android.database.sqlite.SQLiteDatabase

Класс [android.database.sqlite.SQLiteDatabase](#) предназначен для управления Базой Данных SQLite. Содержит методы для добавления, удаления, обновления записей в таблицах Базы Данных (команды SQL: INSERT, UPDATE, DELETE, SELECT), а так же методы, позволяющие создавать и модифицировать структуру Базы Данных (команды SQL: CREATE, ALTER, DROP).

Иерархия класса [android.database.sqlite.SQLiteDatabase](#) имеет следующий вид:

```
java.lang.Object
 |
 +--- android.database.sqlite.SQLiteClosable
 |
 +--- android.database.sqlite.SQLiteDatabase
```

Методы класса [android.database.sqlite.SQLiteDatabase](#):

```
int delete(String table, String whereClause,
           String[] whereArgs);
```

Метод удаляет строку или несколько строк из таблицы Базы Данных. Является аналогом SQL запроса DELETE для удаления строк:

```
DELETE FROM НазваниеТаблицы  
WHERE УсловиеКакиеСтрокиУдалять;
```

Метод **delete()** принимает следующие параметры:

- **String table** — название таблицы, из которой необходимо удалить строку.
- **String whereClause** — содержит условие, определяющее, какие строки необходимо удалить из таблицы. Этот параметр может быть null — для случая, если необходимо удалить все строки из таблицы.
- **String[] whereArgs** — массив значений для параметра **whereClause**. В параметре **whereClause** необходимо использовать символ «?» для указания места, в которое необходимо поместить значение из параметра **whereArgs** (см. Листинг 4.6).

Метод **delete()** возвращает количество строк, которые были удалены.

Давайте рассмотрим несколько примеров применения этого метода. В качестве таблицы для рассматриваемых примеров будем использовать таблицу **Products**, пример создания которой приведена в Листинге 4.1.

В Листинге 4.5 приведены примеры применения метода **delete()** без использования параметра **whereArgs**. В этом случае, значения для условия **whereClause** указаны непосредственно в самом параметре **whereClause**.

**Листинг 4.5.** Пример применения метода delete() объекта android.database.sqlite.SQLiteDatabase для удаления строк

```
MySQLiteOpenHelper dbHelper = new
    MySQLiteOpenHelper(this);
SQLiteDatabase db = dbHelper.getWritableDatabase();

int count = db.delete("Products", "id = 1", null);
Log.d(TAG, "Удалено строк: " + count);

count = db.delete("Products", "name = 'Snickers'", null);
Log.d(TAG, "Удалено строк: " + count);

count = db.delete("Products",
    "price > 20.00 and weight > 50", null);
Log.d(TAG, "Удалено строк: " + count);
```

Однако способ отправки команд в Базу Данных, который приведен в Листинге 4.5 не является удобным, так как значения для условия **whereClause** необходимо помещать непосредственно в саму строку **whereClause**. В Листинге 4.6 приведен пример использования метода **delete()**, в котором значения для условия **whereClause** передаются в параметре **whereArgs**.

**Листинг 4.6.** Пример применения метода delete() объекта android.database.sqlite.SQLiteDatabase для удаления строк

```
MySQLiteOpenHelper dbHelper =
    new MySQLiteOpenHelper(this);
SQLiteDatabase db = dbHelper.getWritableDatabase();

int count = db.delete("Products", "id = ?",
    new String[] { "1" });
Log.d(TAG, "Удалено строк: " + count);
```

```

count = db.delete("Products", "name = ?",
                  new String[] { "Snickers" } );
Log.d(TAG, "Удалено строк: " + count);

count = db.delete("Products", "price > ? and weight > ?",
                  new String[] { "20.00", "50" });
Log.d(TAG, "Удалено строк: " + count);

```

Далее, для того чтобы добавить строку в таблицу Базы Данных, предназначен следующий метод:

```

long insert(String table, String nullColumnHack,
            ContentValues values);

```

Этот метод является аналогом SQL команды INSERT:

**INSERT INTO** ИмяТаблицы **VALUES**  
 (СписокЗначенийСтолбцовДляДобавляемойСтроки);

Метод **insert()** добавляет строку (параметр **values**) в таблицу Базы Данных (параметр **table** — имя таблицы). Параметр **nullColumnHack** является необязательным параметром и может содержать значение null. Этот параметр используется только в одном случае — когда необходимо добавить пустую строку в Базу Данных. В этом случае, параметр **nullColumnHack** должен содержать название столбца таблицы, в который будет явно записано значение null. Именно таким способом SQL позволяет добавлять пустые строки в таблицы.

Метод **insert()** возвращает идентификатор (row ID) добавленной строки или -1 в случае ошибки.

Пример использования метода **insert()** для добавления новой строки в таблицу **Products** приведен в Листинге 4.7.

**Листинг 4.7.** Добавление новой строки в таблицу с помощью метода `insert()` объекта `android.database.sqlite.SQLiteDatabase`

```
MySQLiteOpenHelper dbHelper = new
    MySQLiteOpenHelper(this);
SQLiteDatabase db = dbHelper.getWritableDatabase();

...
ContentValues row = new ContentValues();
row.put("name", "Snickers");
row.put("price", 12.50);
row.put("weight", 45);
long rowID = db.insert("Products", null, row);
```

Далее, для того чтобы внести изменения в строку или в несколько строк таблицы Базы Данных, предназначен следующий метод, который является аналогом SQL команды UPDATE:

```
int update(String table, ContentValues values,
String whereClause, String[] whereArgs);
```

Метод `update()` принимает следующие параметры:

- **String table** — название таблицы Базы Данных, в которой необходимо осуществить обновление строки или нескольких строк.
- **ContentValues values** — содержит значения для обновляемой строки или строк.
- **String whereClause** — содержит условие, какие строки в таблице нужно обновить. Можно использовать специальный символ «?», чтобы указать место, куда будет подставлено значение из массива **whereArgs**.

- `String[] whereArgs` — содержит набор значений для условия `whereClause`. Может быть null.

Метод `update()` возвращает количество строк в таблице Базы Данных, которые были обновлены.

Пример использования метода `update()` для обновления строк таблицы `Products` приведен в Листинге 4.8.

**Листинг 4.8.** Пример использования метода `update()` объекта `android.database.sqlite.SQLiteDatabase` для обновления строки в таблице Базы Данных

```
MySQLiteOpenHelper dbHelper =
        new MySQLiteOpenHelper(this);
SQLiteDatabase db = dbHelper.getWritableDatabase();
...
ContentValues row = new ContentValues();
row.put("name", "Mars");
row.put("price", 14.90);
row.put("weight", 55);
int num = db.update("Products", row, "id = ?",
                    new String[] { "3" });
Log.d(TAG, "Обновлено строк: " + num);
```

Следующий метод предназначен для выборки строк из таблицы Базы Данных:

```
Cursor query(String table,
             String[] columns,
             String whereClause,
             String[] whereArgs,
             String groupBy,
             String having,
             String orderBy);
```

Этот метод является аналогом SQL команды `SELECT`:

```
SELECT ИмяСтолбца1, ИмяСтолбца2, ИмяСтолбцаN
FROM ИмяТаблицы
[WHERE УсловиеКакиеСтрокиВыбирать ]
[GROUP BY ИменаСтолбцовДляГруппированияСтрок
[HAVING УсловиеКакиеГруппыСтрокВыбрать ] ]
[ORDER BY ИменаСтолбцовДляСортировки];
```

Метод **query()** принимает следующие параметры:

- **String table** — имя таблицы Базы Данных, из которой необходимо осуществить выборку строк.
- **String columns** — массив, содержащий имена столбцов, значения которых должны попасть в результат выборки. Если в этот параметр передать значение `null`, то будут выбраны все столбцы таблицы.
- **String whereClause** — условие, определяющее, какие строки необходимо выбрать. Синтаксис условия соответствует синтаксису условия **WHERE** SQL запроса, только без ключевого слова **WHERE**. В строке **whereClause** могут содержаться специальные символы «?», означающие, что в текущую позицию в условии необходимо подставить значение из массива **whereArgs**. Параметр **whereClause** может быть равен `null`. В этом случае будут выбраны все строки таблицы Базы Данных.
- **String whereArgs** — массив, содержащий значения для параметра **whereClause**. Может быть `null`.
- **String groupBy** — содержит фильтр, указывающий как необходимо группировать строки. Синтаксис параметра **groupBy** идентичен синтаксису для оператора **GROUP BY** SQL запроса, только без ключевого слова **GROUP BY**. Параметр может иметь значение `null`.

- **String having** — фильтр, определяющий, какие группы строк (полученные с помощью **GROUP BY**) необходимо оставить в результате выборки. Синтаксис параметра **having** аналогичен синтаксису оператора **HAVING** SQL запроса, только без ключевого слова **HAVING**. Параметр может иметь значение null.
- **String orderBy** — содержит условие, указывающее, как необходимо сортировать строки в результате выборки. Синтаксис параметра **orderBy** аналогичен синтаксису оператора **ORDER BY** SQL запроса, только без ключевого слова **ORDER BY**. Параметр может иметь значение null.

Метод **query()** возвращает объект **android.database.sqlite.SQLiteCursor** (реализующий интерфейс **android.database.Cursor**), который содержит набор строк выбранных из таблицы Базы Данных. Класс **android.database.sqlite.SQLiteCursor** будет рассмотрен в следующем разделе.

Пример применения метода **query()** для выборки строки из таблицы **Products** приведен в Листинге 4.9.

**Листинг 4.9.** Пример применения метода **query()** объекта **android.database.sqlite.SQLiteDatabase** для выборки строк из таблицы Базы Данных

```
Cursor C = db.query("Products",
    newString[] { "name", "price", "weight" },
    "price > ? and weight > ?",
    new String[] { "25.00", "30" },
    null, null, "name");
```

Далее, следующий метод класса **android.database.sqlite.SQLiteDatabase** предназначен для исполнения SQL

запросов, которые не являются запросами SELECT, INSERT, UPDATE, DELETE:

```
void execSQL(String sql);
```

Метод **execSQL()** предназначен для исполнения запросов CREATE, ALTER, DROP. Принимает параметр (**String sql**), который содержит явный запрос SQL.

Пример применения метода **execSQL()** для создания таблицы **Products** (команда **CREATE TABLE** языка SQL) приведен в Листинге 4.10.

**Листинг 4.10.** Пример применения метода **execSQL()** объекта **android.database.sqlite.SQLiteDatabase** для создания таблицы в Базе Данных

```
String query = "CREATE TABLE Products(" +
    "id integer not null primary key autoincrement, " +
    "name text, " + "price real, " +
    "weight integer)";
db.execSQL(query);
```

Следующий метод класса **android.database.sqlite.SQLiteDatabase** предназначен для выполнения многотабличных запросов:

```
Cursor rawQuery(String sql, String[] selectionArgs);
```

Метод **rawQuery()** принимает следующие параметры:

- **String sql** — строка явного SQL запроса. В строке могут располагаться специальные символы «?», которые означают, что в данную позицию в строке необходимо подставить соответствующее значение из массива **selectionArgs**.

- `String[] selectionArgs` — массив, содержащий значения для параметра `sql`.

Метод `rawQuery()` возвращает ссылку на объект `android.database.sqlite.SQLiteDatabase` (реализующий интерфейс `android.database.Cursor`), который содержит результат выборки строк для многотабличного запроса. Пример использования метода `rawQuery()` будет рассмотрен в дальнейших разделах этого урока.

Оставшиеся интересующие нас методы класса `android.database.sqlite.SQLiteDatabase` не требуют такого подробного рассмотрения, как предыдущие, и поэтому мы их рассмотрим вместе:

- `int getVersion()` — возвращает номер версии Базы Данных, к которой подключен объект `android.database.sqlite.SQLiteDatabase`.
- `void setVersion(int version)` — метод устанавливает новое значение для версии Базы Данных, к которой подключен объект `android.database.sqlite.SQLiteDatabase`.
- `final String getPath()` — возвращает файловый путь к файлу Базы Данных, к которой подключен объект `android.database.sqlite.SQLiteDatabase`.
- `static boolean deleteDatabase(File file)` — статический метод. Удаляет файл Базы Данных.

#### 4.4. Класс `android.database.sqlite.SQLiteDatabase`

В предыдущем разделе рассматривались методы объекта `android.database.sqlite.SQLiteDatabase`, которые управляют данными в таблицах Базы Данных. Наиболее важные из этих методов (`query()`, `rawQuery()`) в качестве

результата своего исполнения возвращают объект класса `android.database.sqlite.SQLiteDatabase`, который реализует интерфейс `android.database.Cursor`. В этом разделе мы познакомимся с этим классом.

Объект класса `android.database.sqlite.SQLiteDatabase` содержит набор строк, которые были выбраны из таблицы Базы Данных при помощи объекта `android.database.sqlite.SQLiteDatabase`. Другими словами, он содержит в себе результирующую таблицу, являющуюся результатом выполнения SQL запроса `SELECT` (который был отправлен в СУБД SQLite с помощью методов `query()`, `rawQuery()`).

Иерархия классов для класса `android.database.sqlite.SQLiteDatabase` имеет следующий вид:

```
java.lang.Object
 |
 +-- android.database.AbstractCursor
     |
     +-- android.database.AbstractWindowedCursor
         |
         +-- android.database.sqlite.SQLiteDatabase
```

Объект `android.database.sqlite.SQLiteDatabase` предназначен для чтения строк результирующей таблицы по принципу «Только вперед», то есть данные извлекаются из текущей строки Курсора, затем происходит перемещение к следующей строке для чтения из нее данных и так далее, до тех пор пока не будут перебраны все строки результирующей таблицы, находящейся в Курсоре. Разумеется, класс `android.database.sqlite.SQLiteDatabase` имеет

методы для позиционирования, с помощью которых, при необходимости, можно переместиться к нужной строке и прочитать ее значения заново.

Рассмотрим методы класса `android.database.sqlite.SQLiteDatabase`:

- `void close()` — закрывает объект `android.database.sqlite.SQLiteDatabase`, освобождая при этом все ресурсы, которые были связаны с Курсором.
- `int getColumnIndex(String columnName)` — возвращает индекс столбца из результирующей таблицы, которая находится в объекте `android.database.sqlite.SQLiteDatabase`. Метод очень важен, так как для извлечения значений из столбцов строк используется не имя столбца таблицы, а индекс.
- Тип `getTyp(int columnIndex)` — возвращает данные типа Тип (double, int, long, short и т.д.) из столбца с индексом `columnIndex` текущей строки Курсора.
- `String[] getColumnNames()` — возвращает массив строк, который содержит названия столбцов результирующей таблицы.
- `int getCount()` — возвращает количество строк в результирующей таблице Курсора.
- `SQLiteDatabase getDatabase()` — возвращает объект `android.database.sqlite.SQLiteDatabase`, с которым ассоциирован Курсор.
- `boolean requery()` — исполняет заново запрос, результат исполнения которого привел к созданию текущего Курсора. При этом данные Курсора заполняются заново.

Отдельно рассмотрим методы позиционирования результирующей строки Курсора к нужной строке:

- **boolean moveToFirst()** — перемещает Курсор к первой строке результирующей таблицы. Возвращает признак успешности операции позиционирования.
- **boolean moveToLast()** — перемещает Курсор к последней строке результирующей таблицы. Возвращает признак успешности операции позиционирования.
- **boolean moveToNext()** — перемещает Курсор к следующей строке результирующей таблицы. Возвращает признак успешности операции позиционирования.
- **boolean moveToPosition(int position)** — перемещает Курсор к строке с индексом **position** результирующей таблицы. Возвращает признак успешности операции позиционирования.
- **boolean moveToPrevious()** — перемещает Курсор к предыдущей строке результирующей таблицы. Возвращает признак успешности операции позиционирования.

Пример использования Курсора (объекта **android.database.sqlite.SQLiteDatabase**) для извлечения данных из результирующей таблицы, которая создана в результате исполнения метода **query()** объекта **android.database.sqlite.SQLiteDatabase** приведен в Листинге 4.11.

**Листинг 4.11. Использование объекта  
android.database.sqlite.SQLiteDatabase при извлечении  
данных из результирующей таблицы**

```
MySQLiteOpenHelper dbHelper =  
        new MySQLiteOpenHelper(this);  
SQLiteDatabase db = dbHelper.getWritableDatabase();
```

```

//-- Считывание данных из таблицы Products -----
Cursor C = db.query("Products", null, null,
                    null, null, null, "name");
if (C.moveToFirst())
{
//-- Получение индексов столбцов -----
    int indexId = C.getColumnIndex("id");
    int indexName = C.getColumnIndex("name");
    int indexPrice = C.getColumnIndex("price");
    int indexWeight = C.getColumnIndex("weight");

//-- Извлечение данных из каждой строки
//-- результирующей таблицы
    do
    {
        Log.d(TAG,
            "id = " + C.getInt(indexId) +
            ";name = " + C.getString(indexName) +
            "; price = " + C.getDouble(indexPrice) +
            "; weight = " + C.getInt(indexWeight));

    }
    while (C.moveToNext());
    C.close();
}

```

## 4.5. Пример Android приложения, взаимодействующего с Базой Данных

Теперь, когда мы как следует вооружены полученными знаниями о классах, с помощью которых можно создавать Android-приложения, взаимодействующие с СУБД SQLite, давайте рассмотрим пример такого приложения.

Рассматриваемое в данном разделе приложение будет взаимодействовать с таблицей **Products** из Базы

Данных **MyDbOne**. Базой Данных управляет объект класса **MySQLiteOpenHelper** (см. Листинг 4.2). Весь исходный код приложения данного раздела находится в модуле «app7» среди файлов с исходными кодами, которые прилагаются к этому уроку.

Верстка макета Активности (файл /res/layout/activity\_main.xml) приведена в Листинге 4.12.

#### Листинг 4.12. Xml верстка макета Активности рассматриваемого примера

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.itstep.myapp7.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="10pt"
        android:textColor="#003366"
        android:layout_gravity="center_horizontal"
        android:text="Модуль app7. СУБД SQLite." />

    <ListView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/lvMain" >
    </ListView>

</LinearLayout>
```

Как видно из Листинга 4.12, на Активности находится список `android.widget.ListView` (идентификатор `lvMain`), в котором будет отображаться список продуктов из таблицы **Products** Базы Данных SQLite.

У приложения есть меню, файл ресурсов которого (`/res/menu/menu_main.xml`) приведен в Листинге 4.13. В меню находятся следующие пункты: «Добавить запись в Таблицу», «Удалить запись из Таблицы» и «Редактировать запись». Собственно это и есть основная функциональность рассматриваемого приложения.

**Листинг 4.13.** Файл ресурсов `/res/menu/menu_main.xml`, рассматриваемого в данном разделе приложения

```
<?xml version="1.0" encoding="utf-8"?>

<menu
    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"
    tools:context=".MainActivity">

    <item
        android:id="@+id/action_add"
        android:title="Добавить запись в Таблицу"
        android:orderInCategory="100"
        app:showAsAction="never" />

    <item
        android:id="@+id/action_del"
        android:title="Удалить запись из Таблицы"
        android:orderInCategory="101"
        app:showAsAction="never" />
```

```

<item
    android:id="@+id/action_edt"
    android:title="Редактировать запись"
    android:orderInCategory="102"
    app:showAsAction="never" />
</menu>

```

Внешний вид набора виджетов для элементов списка продуктов, который будет отображаться в виджете `android.widget.ListView` (идентификатор `lvMain`), находится в файле ресурсов `/res/layout/product_item.xml`, содержимое которого представлено в Листинге 4.14.

**Листинг 4.14.** Xml верстка набора виджетов для элементов списка Продуктов, который будет отображаться в виджете `android.widget.ListView`

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="10pt"
        android:layout_gravity="center_horizontal"
        android:layout_margin="2dp"
        android:id="@+id/tvName"
        android:textColor="#0F5119" />

    <LinearLayout
        android:layout_width="match_parent"

```

```
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="8pt"
        android:id="@+id/tvPrice"
        android:gravity="center_horizontal"
        android:layout_weight="1"
        android:textColor="#0F5119" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="8pt"
        android:id="@+id/tvWeight"
        android:gravity="center_horizontal"
        android:layout_weight="1"
        android:textColor="#0F5119" />
</LinearLayout>
</LinearLayout>
```

Внешний вид Активности виджета **android.widget.ListView** (идентификатор **lvMain**) с элементами и меню приложения изображен на Рис. 4.1.

Для подключения к Базе Данных **MyDbOne** используется класс **MySQLiteOpenHelper** из Листинга 4.2. Единственное, что пришлось добавить в этот класс, так это добавление в таблицу **Products** нескольких записей (в методе **onCreate()**), чтобы при первом запуске нашего приложения таблица не была пустой, и соответственно, чтобы список **lvMain** также не был пустым. Измененный код метода **onCreate()** класса **MySQLiteOpenHelper** представлен в Листинге 4.15.

**Листинг 4.15.** Создание таблицы Products в Базе Данных



**Рис. 4.1.** Внешний вид Активности, списка android.widget.ListView и меню приложения из Листингов 4.12, 4.13, 4.14

MyDbOne и заполнение этой таблицы несколькими начальными записями в методе обратного вызова onCreate() класса MySQLiteOpenHelper

```
@Override
public void onCreate (SQLiteDatabase db)
{
    Log.d(TAG, "onCreate: " + db.getPath());
    /* ----- Создание таблиц ----- */
    //--- Создание таблицы Products ---
    String query = "CREATE TABLE Products(" +
        "id integer not null primary key autoincrement, " +
        "name text, " + "price real, " +
        "weight integer)";
    db.execSQL(query);
    //--- Заполнение таблицы несколькими продуктами --
    ContentValues row = new ContentValues();
    row.put("name", "Snickers");
    db.insert("Products", null, row);
}
```

```
        row.put("price", 12.50);
        row.put("weight", 45);
        db.insert(MySQLiteOpenHelper.tblNameProducts,
                  null, row);

        row = new ContentValues();
        row.put("name", "Mars");
        row.put("price", 13.90);
        row.put("weight", 50);
        db.insert(MySQLiteOpenHelper.tblNameProducts,
                  null, row);

        row = new ContentValues();
        row.put("name", "Bounty");
        row.put("price", 15.30);
        row.put("weight", 60);
        db.insert(MySQLiteOpenHelper.tblNameProducts,
                  null, row);
    }
```

Далее, для отображения списка Продуктов, извлеченных из Базы Данных, понадобится Адаптер Данных, например **android.widget.ArrayAdapter<T>**. Следовательно, нужен класс, который будет представлять элемент списка для адаптера Данных. Класс **Product** приведен в Листинге 4.16.

**Листинг 4.16.** Класс **Product** представляет один элемент списка для Адаптера Данных и содержит данные одной строки из таблицы **Products** Базы Данных

```
 /**
 * Class Product – Представляет одну запись из
 * таблицы Базы Данных Products.
 */
class Product
{
```

```
//-- Class members -----
/***
 * Название Продукта
 */
public String name;

/***
 * Стоимость Продукта
 */
public double price;

/***
 * Вес Продукта
 */
public int weight;

/***
 * Идентификатор строки Продукта в таблице Products
 * Базы Данных
 */
public int id;

//-- Class methods -----
public Product(String name, double price,
               int weight, int id)
{
    this.name = name;
    this.price = price;
    this.weight = weight;
    this.id = id;
}

@Override
public String toString()
{
    return "" + this.id + ": " + this.name +
           ": " + this.price + ": " + this.weight;
}
}
```

Код заполнения списка `android.widget.ListView` (идентификатор `lvMain`) осуществляется при старте Активности в методе `onCreate()` класса Активности и приведен в Листинге 4.17.

**Листинг 4.17.** Считывание продуктов из таблицы Products  
Базы Данных и заполнение ими списка `android.widget.ListView`  
через объект Адаптера Данных `android.widget.ArrayAdapter<T>`

```
/**  
 * Class MainActivity – Главная Активность приложения  
 * -----  
 */  
public class MainActivity extends  
AppCompatActivity  
{  
    //--- Class members -----  
    /**  
     * Объект MySQLOpenHelper для управлением подключения  
     * к Базе Данных и для управления версиями Базу Данных.  
     */  
    private MySQLiteOpenHelper dbHelper;  
  
    /**  
     * Адаптер данных для отображения в списке  
     * android.widget.ListView.  
     */  
    private ArrayAdapter<Product> adapter;  
  
    ...  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState)  
    {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        //--- DBHelper -----  
        this.dbHelper = new MySQLiteOpenHelper(this);  
    }  
}
```

```
//-- ListView -----
    ListView LV = (ListView) this.findViewById(
        R.id.lvMain);

    ArrayList<Product> lstProducts =
        new ArrayList<>();

/*
 * Считывание данных из БД и заполнение ListView
 */
final SQLiteDatabase db = this dbHelper.
    getWritableDatabase();
Cursor C = db.query(MySQLiteOpenHelper.
    tblNameProducts, null, null, null,
    null, null, "name");

//-- Извлечение данных из
//-- android.database.sqlite.SQLiteDatabase -----
if (C.moveToFirst())
{
//-- Получение индексов столбцов -----
    int indexId = C.getColumnIndex("id");
    int indexName = C.getColumnIndex("name");
    int indexPrice = C.getColumnIndex("price");
    int indexWeight = C.getColumnIndex("weight");

//-- Перебор всех строк в Курсоре -----
    do
    {
        lstProducts.add(new Product
            (C.getString(indexName),
            C.getDouble(indexPrice),
            C.getInt(indexWeight),
            C.getInt(indexId))
        );
    }
}
```

```
        while (C.moveToFirst());  
  
        C.close();  
    }  
    else  
        Log.d(TAG, "Невозможно позиционироваться  
на первую строку Курсора");  
  
/*  
 * ArrayAdapter для android.widget.ListView  
 * -----  
 */  
//--- Создание ArrayAdapter для ListView -----  
this.adapter = new ArrayAdapter<Product>(this,  
    R.layout.product_item, R.id.tvName,  
    lstProducts);  
  
LV.setAdapter(adapter);  
}  
...  
}
```

Добавим, что в рассматриваемом примере используется custom набор виджетов для отображения и подсветки выбранных элементов списка **android.widget.ListView** (см. Рис. 4.1). Этот код не отображен в Листинге 4.17, но с ним можно ознакомиться в модуле «app7» проекта среди файлов исходного кода, которые прилагаются к этому уроку. Сам механизм организации custom отображения и подсветки выбранных элементов списка **android.widget.ListView** рассматривался в одном из прошлых уроков данного курса. Отметим только, что индекс текущего выделенного элемента списка Продуктов **android.widget.ListView** хранится в поле объекта Активности

с названием **curItem**. Объявление этого поля представлено в Листинге 4.18. Это поле будет часто использоваться в рассматриваемом примере.

**Листинг 4.18.** Объявление поля **curItem** класса Активности, которое используется в механизме **custom** отображения элементов списка **android.widget.ListView** и хранит индекс текущего отмеченного элемента списка Продуктов

```
/**  
 * Class MainActivity –  
 * Главная Активность приложения  
 * -----  
 */  
public class MainActivity extends AppCompatActivity  
{  
    ...  
    /**  
     * Индекс выбранного элемента списка  
     */  
    private int curItem = -1;  
    ...  
}
```

Далее, для добавления и редактирования данных Продукта, необходимо Диалоговое Окно, в которое пользователь мог бы вводить данные. Для создания Диалогового Окна используются классы **android.app.AlertDialog** (само Диалоговое окно) и **android.app.AlertDialog.Builder** (объект построитель Диалоговых Окон). Для рассматриваемого примера было выбрано решение, в котором Диалоговое Окно **android.app.AlertDialog** создается один раз при старте Активности в методе **onCreate()** и в дальнейшем это Диалоговое Окно только отображается при выборе

пользователем пунктов меню «Добавить новую запись в Таблицу» / «Редактировать запись».

Для содержимого Диалогового окна **android.app.AlertDialog** создан файл ресурсов `/res/layout/dialog_add_item.xml` с xml версткой внешнего вида набора виджетов. Содержимое файла `/res/layout/dialog_add_item.xml` приведено в Листинге 4.19.

**Листинг 4.19.** Xml верстка набора виджетов для содержимого Диалогового Окна `android.app.AlertDialog`, используемое для функциональности «Добавить новую запись в Таблицу» / «Редактировать запись» рассматриваемого примера

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="*">

    <TableRow>
        <TextView android:text="Название"
            android:layout_gravity=
                "right|center_vertical"/>

        <EditText
            android:id="@+id/edtName" />
    </TableRow>

    <TableRow>
        <TextView android:text="Цена"
            android:layout_gravity=
                "right|center_vertical"/>
```

```
<EditText  
    android:id="@+id edtPrice" />  
</TableRow>  
  
<TableRow>  
    <TextView android:text="Bec"  
        android:layout_gravity=  
            "right|center_vertical"/>  
  
    <EditText  
        android:id="@+id edtWeight" />  
</TableRow>  
  
<TableRow>  
    <TextView  
        android:id="@+id tvHidden"  
        android:layout_span="2"  
        android:visibility="invisible"  
        android:layout_height="0dp" />  
</TableRow>  
  
</TableLayout>
```

Вид Диалогового окна `android.app.AlertDialog` из Листинга 4.19 можно увидеть на Рис. 4.2, 4.3.

Как видно из Листинга 4.19, для содержимого Диалогового Окна `android.app.AlertDialog` созданы текстовые поля `android.widget.EditText` (идентификаторы `edtName`, `edtPrice`, `edtWeight`), в которые пользователь сможет водить данные для Названия, Стоимости и Веса продукта соответственно. В Листинге 4.19 жирным шрифтом выделен код создания «невидимого» виджета `android.`

`widget.TextView` (идентификатор `tvHidden`), которое предназначено для размещения идентификатора строки таблицы Продуктов (`Products`) Базы Данных.

Дело в том, что Диалоговое Окно будет использовано как для добавления нового Продукта в Базу Данных, так и для редактирования уже существующего Продукта в Базе Данных. Чтобы в обработчике события нажатия на кнопку «Применить» Диалогового Окна `android.app.AlertDialog` можно было определить, что необходимо сделать с данными, которые пользователь ввел в окне — Добавить в таблицу `Products` Базы Данных новую строку или заменить уже существующую, — в этом скрытом поле будет хранится признак операции «Добавит» или «Заменить». То есть, если в этом скрытом поле будет хранится пустое значение, то значит необходимо добавить (`INSERT`) данные в таблицу `Products` Базы Данных, в противном случае, в этом поле будет храниться идентификатор строки таблицы `Products`, которую необходимо обновить (`UPDATE`) теми данными, которые пользователь ввел в окно. Разумеется, для хранения идентификатора строки таблицы можно было бы использовать, например свойство `Tag` любого из виджетов Диалогового Окна, что было бы правильно. Тем не менее, способ используемый в Листинге 4.19, также имеет право на существование.

Создание Диалогового Окна `android.app.AlertDialog` с обработчиками события нажатия на кнопки «Применить» и «Отменить» осуществляется в методе `onCreate()` класса главной Активности и приведено в Листинге 4.20.

**Листинг 4.20.** Создание Диалогового Окна  
для добавления / редактирования записей таблицы  
Products Базы Данных

```
/**  
 * Class MainActivity – Главная Активность приложения  
 * -----  
 */  
public class MainActivity extends AppCompatActivity  
{  
    ...  
    /**  
     * Диалоговое окно для  
     * ввода / редактирования данных Продукта  
     */  
    private AlertDialog dialog;  
  
    //--- Class methods -----  
    @Override  
    protected void onCreate(Bundle savedInstanceState)  
{  
    ...  
    /*  
     * Создание объекта Диалогового окна  
     * android.app.AlertDialog для добавления /  
     * редактирования данных Продукта  
     */  
    //--- AlertDialog.Builder – объект построитель  
    //--- Диалоговых окон -----  
    AlertDialog.Builder builder =  
        new AlertDialog.Builder(  
            this, AlertDialog.THEME_HOLO_LIGHT);  
  
    //--- Обработчик кнопки "Отменить" -----  
    builder.setNegativeButton("Отменить",  
        new DialogInterface.OnClickListener()
```



```

    {
        throw new Exception(
            "Заполнены не все поля!");
    }

    double productPrice = Double.
        parseDouble(productPriceStr);
    int productWeight = Integer.
        parseInt(productWeightStr);

    //--- Занесение в Базу Данных нового товара -----
    ContentValues row = new ContentValues();
    row.put("name", productNameStr);
    row.put("price", productPrice);
    row.put("weight", productWeight);

    //--- Определение что необходимо сделать -
    //--- добавить строку или заменить
    String strHiddenId = ((TextView)
        (AlertDialog) dialog).
        findViewById(R.id.tvHidden)).
        getText().toString();
    if (strHiddenId.isEmpty())
    {
        //--- Добавление продукта -----
        long rowID = db.insert(
            MySQLiteOpenHelper.
            tblNameProducts, null, row);

        if (rowID == -1)
            throw new Exception(
                "Строка не добавлена в таблицу
                (row ID = -1)");
    }

    Toast.makeText(MainActivity.this,
        "Продукт успешно добавлен",
        Toast.LENGTH_SHORT).show();
}

```

```
//-- Добавление в ListView нового продукта -----
    MainActivity.this.adapter.add(
        new Product(productNameStr,
        productPrice, productWeight,
        (int) rowID));
    }
else
{
//-- Обновление продукта -----
    int cnt = db.update(
        MySQLiteOpenHelper.
        tblNameProducts, row, "id=?",
        new String[] { strHiddenId });

    Toast.makeText(MainActivity.this,
        "Обновлено строк: " + cnt,
        Toast.LENGTH_SHORT).show();

    if (cnt > 0)
    {
//-- Внесение изменений в список ListView -----
        Product P =
            MainActivity.this.adapter.
            getItem(MainActivity.this.
            curItem);
        P.name = productNameStr;
        P.price = productPrice;
        P.weight = productWeight;

        MainActivity.this.adapter.remove(P);
        MainActivity.this.adapter.insert(P,
            MainActivity.this.curItem);
    }
}
catch (Exception e)
{
```

```
        Toast.makeText(MainActivity.this,
                "Данные введены неправильно: " +
                e.getMessage(),
                Toast.LENGTH_SHORT).show();
            Log.d(TAG, e.getMessage());
        }
    }
});  
  
//-- Диалоговое окно для ввода данных -----
this.dialog = builder.create();
this.dialog.setView(this.getLayoutInflater().
        inflate(R.layout.dialog_add_item, null, true));
...
}
```

В Листинге 4.20 самая главная функциональность находится в методе обработчике события нажатия на кнопку «Применить» — метод **onClick()** объекта **DialogInterface.OnClickListener**. Объявление метода **onClick()** в Листинге 4.20 выделено жирным шрифтом. В этом методе происходит считывание введенных пользователем данных (Название, Стоимость, вес продукта) из текстовых полей **android.widget.EditText** (идентификаторы **edtName**, **edtPrice**, **edtWeight**) и если значения в этих текстовых полях не пустые, то считывается значение из «скрытого» текстового поля **android.widget.TextView** (идентификатор **tvHidden**), которое содержит или пустую строку в случае, если необходимо выполнить добавление новой записи в таблицу Продуктов (**Products**) Базы Данных или содержит значение идентификатора строки таблицы **Products** в случае, если необходимо выполнить обновление записи в таблице **Products**. Откуда в этом скрытом поле появляются такие

данные, можно увидеть в Листинге 4.21. Также в Листинге 4.20 выделены жирным шрифтом вызовы методов `insert()` и `update()` объекта `android.database.sqlite.SQLiteDatabase`, с помощью которых и осуществляется внесение изменений в Базу Данных.

Диалоговое Окно `android.app.AlertDialog`, представленное в Листинге 4.20, вызывается в обработчике событий выбора пунктов меню приложения в методе `onOptionsItemSelected()` класса Активности. Код метода `onOptionsItemSelected()` приведен в Листинге 4.21.

**Листинг 4.21.** Обработка событий выбора пунктов меню приложения и отображение Диалогового Окна `android.app.AlertDialog` для ввода данных

```
@Override
public boolean onOptionsItemSelected(MenuItem item)
{
    //-- Получение доступа к БД -----
    final SQLiteDatabase db = this.dbHelper.
        getWritableDatabase();

    //-- Обработка выбранного пункта меню -----
    int id = item.getItemId();
    switch (id)
    {
        /*
         * Добавление записи в БД
         * -----
         */
        case R.id.action_add:
        {
            //-- Диалоговое окно "Добавить новый продукт" ---
            this.dialog.setTitle("Добавить новый Продукт");
            //-- Показываем Диалоговое окно -----
            this.dialog.show();
        }
    }
}
```

```

//-- Очистка текстовых полей Диалогового окна ---
    if (this.dialog.findViewById(R.
        id.edtName) != null)
    {
        ((EditText) this.dialog.findViewById(
            R.id.edtName )).setText("");
        ((EditText) this.dialog.findViewById(
            R.id.edtPrice )).setText("");
        ((EditText) this.dialog.findViewById(
            R.id.edtWeight)).setText("");
        ((TextView) this.dialog.findViewById(
            R.id.tvHidden )).setText("");
    }
}
break;
/*
 * Удаление записи из БД
 * -----
 */
case R.id.action_del:
//-- Находим текущий выбранный продукт -----
    if (this.curItem == -1)
    {
        Toast.makeText(this, "Продукт не выбран",
            Toast.LENGTH_SHORT).show();
        break;
    }

//-- Находим текущий выделенный элемент списка --
    Product P = this.adapter.getItem(this.
        curItem);

//-- Удаляем из таблицы Базы Данных -----
    int rowAffected = db.delete(
        MySQLiteOpenHelper.tblNameProducts,
        "id=?", new String[] { String.
            valueOf(P.id) });

```

```
        Toast.makeText(this,
                "Удалено строк: " + rowAffected,
                Toast.LENGTH_SHORT).show();

//-- Внесение изменений в список ListView (lvMain) --
        this.adapter.remove(P);
        break;

/*
 *   Обновление записи из БД
 *
 */
        case R.id.action_edt:
//-- Диалоговое окно "Редактировать Продукт" -----
        this.dialog.setTitle(
                "Редактировать Продукт");

//-- Находим текущий выбранный продукт -----
        if (this.curItem == -1)
        {
            Toast.makeText(this, "Продукт не выбран",
                    Toast.LENGTH_SHORT).show();
            break;
        }
        Product P1 = this.adapter.getItem(this.
                curItem);

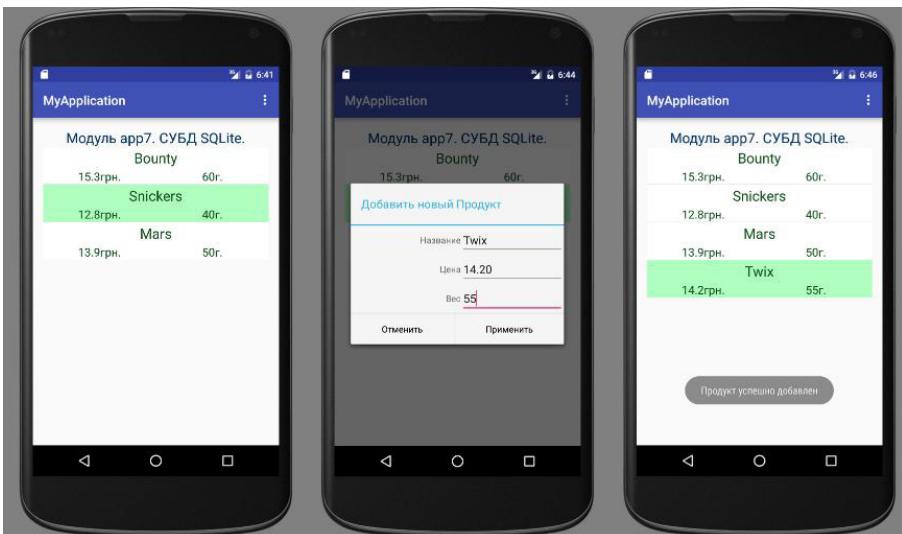
//-- Показываем Диалоговое окно -----
        this.dialog.show();

//-- Инициализация текстовых полей Диалогового окна
        if (this.dialog.findViewById(
                R.id.edtName) != null)
        {
            ((EditText) this.dialog.findViewById(
                R.id.edtName)).setText(P1.name);
```

```
        ((EditText) this.dialog.findViewById( R.id.edtPrice)).setText(String. valueOf(P1.price));  
        ((EditText) this.dialog.findViewById( R.id.edtWeight)).setText(String. valueOf(P1.weight));  
        ((TextView) this.dialog.findViewById( R.id.tvHidden )).setText(String. valueOf(P1.id));  
    }  
    break;  
}  
return super.onOptionsItemSelected(item);  
}
```

Посмотрим, что происходит в Листинге 4.21. Когда пользователь выбирает пункт меню «Добавить новую запись в Таблицу» (**R.id.action\_add**), то в заголовок Диалогового Окна помещается текст «Добавить новый Продукт» и содержимое всех текстовых полей **android.widget.EditText** (идентификаторы **edtName**, **edtPrice**, **edtWeight**) очищается. Также очищается значение «скрытого» поля **android.widget.TextView** (идентификатор **tvHidden**) как признак того, что при закрытии Диалогового Окна по кнопке «Применить» необходимо будет выполнить операцию добавления новой записи в таблицу **Products**.

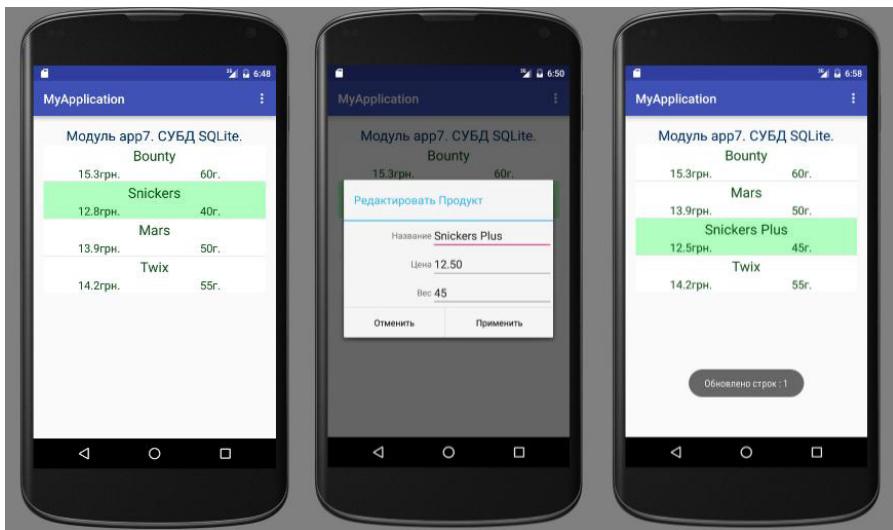
В случае, когда пользователь выбирает пункт меню «Удалить запись из Таблицы» (**R.id.action\_del**), то извлекается ссылка на Продукт из Адаптера Данных (поле **adapter** класса Активности) с индексом из поля **curItem** (см. Листинг 4.18) и происходит удаление записи из таблицы **Products** при помощи вызова метода **delete()** объекта



**Рис. 4.2.** Внешний вид работы примера из Листингов 4.20 и 4.21 для добавления нового Продукта в Базу Данных

`android.database.sqlite.SQLiteDatabase`. Методу `delete()` в качестве условия для удаления записи передается идентификатор строки выбранного Продукта (в Листинге 4.21 этот фрагмент кода выделен жирным шрифтом).

В случае, когда пользователь выбирает пункт меню «Редактировать запись» (`R.id.action_edt`), то в заголовок Диалогового Окна помещается текст «Редактировать Продукт» и извлекается ссылка на Продукт из Адаптера Данных (поле `adapter` класса Активности) с индексом из поля `curItem` (см. Листинг 4.18) для того, чтобы разместить в текстовые поля `android.widget.EditText` (идентификаторы `edtName`, `edtPrice`, `edtWeight`) Диалогового Окна значения для редактирования выбранного Продукта. Также в «скрытое» поле `tvHidden` записывается идентификатор строки в таблице `Products` как признак того, что при закрытии Диалогового



**Рис. 4.3.** Внешний вид работы примера из Листингов 4.20 и 4.21 для редактирования Продукта в Базе Данных

Окна по кнопке «Применить» необходимо будет выполнить обновление редактируемой записи в таблице.

Внешний вид работы примера из Листинга 4.21 для операции добавления нового Продукта изображен на Рис. 4.2.

Внешний вид работы примера из Листинга 4.21 для операции редактирования Продукта изображен на Рис. 4.3.

Напомним, что весь исходный код рассматриваемого в данном разделе примера находится в модуле «app7» проекта с исходными кодами, которые прилагаются этому уроку.

Теперь давайте посмотрим на то, что делается в программном коде из Листингов 4.20 и 4.21 со списком Продуктов в виджете **android.widget.ListView** (идентификатор **lvMain**), после внесения изменений в Базу Данных. Так вот, после внесения изменений (в таблицу **Products**), программе приходится самостоятельно обновлять содержимое списка

**android.widget.ListView** (идентификатор lvMain). Эти фрагменты кода (для вызовов методов **insert()**, **update()**, **delete()**) еще раз приведены в Листингах 4.22, 4.23, 4.24 соответственно. «Ручное» обновление списка **android.widget.ListView** (через обновление Адаптера Данных) в этих Листингах выделено жирным шрифтом.

**Листинг 4.22.** После вызова метода **insert()** для добавления строки в таблицу Базы Данных происходит добавление нового элемента в список **android.widget.ListView**

```
//-- Добавление продукта -----
long rowID = db.insert(MySQLiteOpenHelper.
                      tblNameProducts, null, row);
if (rowID == -1)
    throw new Exception("Строка не добавлена
                         в таблицу (row ID = -1)");
Toast.makeText(MainActivity.this,
              "Продукт успешно добавлен",
              Toast.LENGTH_SHORT).show();

//-- Добавление в ListView нового продукта -----
MainActivity.this.adapter.add(
    new Product(productNameStr,
                productPrice, productWeight, (int) rowID));
```

**Листинг 4.23.** После вызова метода **update()** для обновления строки в таблице Базы Данных происходит обновление соответствующего элемента в списке **android.widget.ListView**

```
//-- Обновление продукта -----
int cnt = db.update(MySQLiteOpenHelper.
                      tblNameProducts, row, "id=?",
                      new String[] { strHiddenId });
Toast.makeText(MainActivity.this, "Обновлено строк: " +
cnt, Toast.LENGTH_SHORT).show();
```

```

if (cnt > 0)
{
//-- Внесение изменений в список ListView -----
Product P = MainActivity.this.adapter.getItem(
    MainActivity.this.curItem);
P.name = productNameStr;
P.price = productPrice;
P.weight = productWeight;

MainActivity.this.adapter.remove(P);
MainActivity.this.adapter.insert(P,
        MainActivity.this.curItem);
}

```

**Листинг 4.24.** После вызова метода `delete()` для удаления строки из таблицы `Products` Базы Данных происходит удаление соответствующего элемента из списка `android.widget.ListView`

```

//-- Находим текущий выделенный элемент списка -----
Product P = this.adapter.getItem(this.curItem);
//-- Удаляем из таблицы Базы Данных -----
int rowAffected = db.delete(
    MySQLiteOpenHelper.tblNameProducts,
    "id=?", new String[] { String.valueOf(P.id) });
Toast.makeText(this, "Удалено строк: " +
    rowAffected, Toast.LENGTH_SHORT).show();
//-- Внесение изменений в список ListView (lvMain) --
this.adapter.remove(P);

```

Посмотрев на Листинги 4.22, 4.23, 4.24, возникает вопрос — а нет ли специального инструмента (например какого-нибудь Адаптера Данных), который бы избавил бы разработчика от «ручной» работы по поддержанию соответствия содержимого таблицы Базы Данных и списка `android.widget.ListView`? Конечно, такой инструмент есть

и им, конечно же, являются Адаптеры Данных. Для взаимодействия с Базой Данных существуют следующие классы Адаптеров Данных: абстрактный класс [android.widget.CursorAdapter](#), и производные от него классы [android.widget.ResourceCursorAdapter](#) и [android.widget.SimpleCursorAdapter](#). Класс [android.widget.ResourceCursorAdapter](#) больше подходит для отображения результирующих таблиц состоящих из одного столбца. В нашем случае, когда отображается список Продуктов и каждый Продукт состоит из нескольких полей («Название», «Стоимость» «Вес»), лучше всего подойдет класс [android.widget.SimpleCursorAdapter](#). Именно этот класс и рассматривается в следующем разделе.

#### **4.6. Адаптер Данных, класс [android.widget.SimpleCursorAdapter](#)**

Класс [android.widget.SimpleCursorAdapter](#) является простым Адаптером Данным для отображения столбцов из Курсора (например из объекта [android.database.sqlite.SQLiteDatabase](#)) в виджетах [android.widget.TextView](#) и [android.widget.ImageView](#), которые объявлены в xml макетах.

Иерархия классов для класса [android.widget.SimpleCursorAdapter](#) имеет следующий вид:

```
java.lang.Object
|
+- android.widget.BaseAdapter
|
+- android.widget.CursorAdapter
|
+- android.widget.ResourceCursorAdapter
|
+- android.widget.SimpleCursorAdapter
```

Объект `android.widget.SimpleCursorAdapter` очень сильно похож Адаптер Данных `android.widget.SimpleAdapter`, который рассматривался в предыдущих уроках этого курса. Следовательно, разобраться с Адаптером Данных `android.widget.SimpleCursorAdapter` нам будет проще. Одна важная особенность Адаптера Данных `android.widget.SimpleCursorAdapter` заключается в том, что ему передается Курсор (объект, реализующий интерфейс `android.database.Cursor`, например `android.database.sqlite.SQLiteDatabase`), который и служит источником данных для Адаптера.

**Внимание!** *Важно! Для работы с объектом android.widget.SimpleAdapter столбец таблицы первичного ключа таблицы должен называться «`_id`».*

Создать объект класса `android.widget.SimpleCursorAdapter` можно с помощью конструктора:

```
SimpleCursorAdapter (Context context, int layout,  
                     Cursor c, String[] from, int[] to, int flags);
```

Конструктор класса `android.widget.SimpleCursorAdapter` принимает следующие параметры:

- **Context context** — ссылка на объект Контекста приложения.
- **int layout** — идентификатор файла ресурсов, который содержит xml макет набора виджетов, которые представляют элемент списка `android.widget.ListView`.
- **Cursor C** — объект `android.database.Cursor` (`android.database.sqlite.SQLiteDatabase`), который содержит результатирующую таблицу (выборку из Таблицы Базы

Данных), которую необходимо отобразить в списке `android.widget.ListView`.

- `String[] from` — строковый массив, содержащий названия столбцов результирующей таблицы, значения которых необходимо отображать в списке `android.widget.ListView`.
- `int[] to` — массив идентификаторов виджетов из xml макета набора виджетов (идентификатор `layout`), которые соответствуют элементам массива `from`. Значение из столбца, имя которого указано в элементе массива `from`, необходимо отобразить в виджете с идентификатором из соответствующего элемента массива `to`.
- `int flags` — набор дополнительных флагов. На текущий момент действителен только один флаг `Cursor.FLAG_REGISTER_CONTENT_OBSERVER`, который указывает на необходимость вызова метода `onContentChanged()` Адаптера Данных, когда Курсор получает уведомления об изменении.

Методов добавления и удаления записей в классе `android.widget.SimpleCursorAdapter` нет. Это логично. Ведь изменения необходимо вносить в Базу Данных, а не в Адаптер Данных. Поэтому, после создания объекта `android.widget.SimpleAdapter`, чаще всего придется пользоваться следующими методами:

- `Cursor swapCursor(Cursor c)` — устанавливает в объект Адаптера Данных `android.widget.SimpleCursorAdapter` новый Курсор (параметр `c`). Предыдущий Курсор, который был установлен в Адаптере Данных, возвращается этим методом. Возвращенный предыдущий

Курсор рекомендуется закрыть с помощью вызова метода `close()`, освободив тем самым все ресурсы, занимаемые предыдущим Курсором.

- **Cursor getCursor()** — возвращает ссылку на текущий Курсор, который назначен Адаптеру Данных `android.widget.SimpleCursorAdapter`.

Теперь можно смело переходить к рассмотрению примера применения объекта Адаптера Данных `android.widget.SimpleCursorAdapter`. Для этого примера создан модуль «app8» в проекте с файлами исходного кода, которые прилагаются к этому уроку. Пример будет так же взаимодействовать с таблицей Продуктов (**Products**) Базы Данных. Внешний вид и функциональность для пользователя примера этого раздела ничем не будет отличаться от внешнего вида и функциональности из примера предыдущего раздела (см. Рис 4.2 и 4.3). Это позволит нам сконцентрироваться только над изменениями, которые дает использование объекта `android.widget.SimpleCursorAdapter`.

Первое изменение, которое будет отличать текущий пример от примера из предыдущего раздела, заключается в том, что пропала необходимость в классе **Product** из Листинга 4.16 (который содержал в себе данные одной строки из таблицы Продуктов). Это произошло потому, что записи, представляющие элементы списка, будут теперь находиться в Курсоре, который будет назначен Адаптеру Данных `android.widget.SimpleCursorAdapter`.

Следующее изменение касается класса **MySQLiteOpenHelper** (который перекочевал из предыдущего примера в текущий). В классе **MySQLiteOpenHelper** объявлены строковые константы, содержащие названия столбцов

таблицы **Products**. Это сделано с целью избежать ошибок в программе, которые связаны с опечатками в названиях столбцов. Также, пример текущего раздела будет работать с Базой Данных **MyDbTwo**. Объявление строковых констант в классе **MySQLiteOpenHelper** приведено в Листинге 4.25. Все остальные методы класса **MySQLiteOpenHelper** такие же, как в Листинге 4.2.

**Листинг 4.25.** Объявление строковых констант для названий столбцов таблицы **Products** с целью избежания опечаток в названиях столбцов в программном коде

```
/**  
 * Class MySQLiteOpenHelper  
 * -----  
 */  
  
class MySQLiteOpenHelper extends SQLiteOpenHelper  
{  
    //--- Class constants -----  
    /**  
     * Tag для Log.d  
     */  
    private final static String TAG = "===== MySQLite";  
  
    /**  
     * Название Базы Данных  
     */  
    private final static String dbName = "MyDbTwo";  
  
    /**  
     * Версия Базы Данных  
     */  
    private final static int dbVersion = 1;
```

```
/**  
 * Название таблицы Продуктов (Products) в Базе Данных  
 */  
public final static String tblNameProducts =  
        "Products";  
  
/**  
 * Имя столбца "name" таблицы Продуктов (Products)  
 */  
public final static String colProductName = "name";  
  
/**  
 * Имя столбца "price" таблицы Продуктов (Products)  
 */  
public final static String colProductPrice = "price";  
  
/**  
 * Имя столбца "weight" таблицы Продуктов (Products)  
 */  
public final static String colProductWeight = "weight";  
  
/**  
 * Имя столбца "_id" таблицы Продуктов (Products)  
 */  
public final static String colId = "_id";  
  
...  
}
```

Для рассматриваемого примера, так же как и в предыдущем примере, будет реализована custom подсветка выбранных элементов списка, и, так же, пользовательский ввод данных будет осуществляться с помощью Диалогового Окна **android.app.AlertDialog**. Меню приложения тоже оставим без изменения (см. Листинг 4.13). Объявление

полей класса Активности текущего примера приведено в Листинге 4.26.

**Листинг 4.26.** Поля класса главной Активности текущего примера

```
public class MainActivity extends AppCompatActivity
{
    //--- Class members -----
    /**
     * Объект MySQLOpenHelper для управлением
     * подключения к Базе Данных и для управления
     * версиями Базу Данных.
     */
    private MySQLiteOpenHelper dbHelper;

    /**
     * Адаптер данных для android.widget.ListView
     */
    private SimpleCursorAdapter adapter;

    /**
     * Цвет фона не выбранного элемента списка
     */
    private int nrmlColor = Color.rgb(0xFF, 0xFF, 0xFF);

    /**
     * Цвет фона выбранного элемента списка
     */
    private int slctColor = Color.rgb(0xB0, 0xFF, 0xC0);

    /**
     * Индекс выбранного элемента списка
     */
    private int curItem = -1;

    /**
     * Ссылка на виджет текущего выбранного элемента
     * списка
    
```

```

    */
private View curView = null;

/**
 * Диалоговое окно для
 * ввода / редактирования данных Продукта
 */
private AlertDialog dialog;

...
}

```

Теперь рассмотрим создание Адаптера Данных **android.widget.SimpleCursorAdapter** (поле **adapter** Активности). Этот объект создается в методе **onCreate()** класса Активности и программный код его создания приведен в Листинге 4.27.

#### **Листинг 4.27.** Создание объекта Адаптера Данных **android.widget.SimpleCursorAdapter**

```

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

    //--- DBHelper -----
    this.dbHelper = new MySQLiteOpenHelper(this);
    /*
     *
     * Считывание данных из БД и заполнение ими ListView
     *
     */
    final SQLiteDatabase db = this.dbHelper.
        getWritableDatabase();

```

```
Cursor C = db.query(MySQLiteOpenHelper.  
    tblNameProducts, null, null, null, null,  
    null, MySQLiteOpenHelper.colProductName);  
  
//--- Получение ссылки на ListView (lvMain) -----  
ListView LV = (ListView) this.findViewById(  
    R.id.lvMain);  
/*  
 * SimpleCursorAdapter (Адаптер данных) для ListView  
 * -----  
 */  
this.adapter = new SimpleCursorAdapter(this,  
    R.layout.product_item, C, new String[]  
{  
    MySQLiteOpenHelper.colProductName,  
    MySQLiteOpenHelper.colProductPrice,  
    MySQLiteOpenHelper.colProductWeight  
},  
new int[]  
{  
    R.id.tvName,  
    R.id.tvPrice,  
    R.id.tvWeight  
},  
0)  
{  
    @Override  
    public View getView(int position,  
        View convertView, ViewGroup parent)  
    {  
        View view = super.getView(position,  
            convertView, parent);  
  
    //--- Подсветка отмеченного элемента списка -----  
    if (position == MainActivity.this.curItem)  
    {
```

```
        view.setBackgroundColor(MainActivity.  
        this.slctColor);  
    }  
  
    else  
    {  
        view.setBackgroundColor(MainActivity.  
        this.nrmlColor);  
    }  
  
    //--- Возвращаем виджет, представляющий  
    //--- элемент списка -----  
    return view;  
}  
};  
LV.setAdapter(this.adapter);  
  
..  
}
```

Также в методе **onCreate()** класса Активности приложения создается Диалоговое Окно **android.app.AlertDialog**, с помощью которого пользователь будет вводить данные для добавляемых или обновляемых Продуктах (поле **dialog** Активности). При создании, Диалоговому окну сразу же назначаются обработчики нажатия на кнопки «Применить» и «Отменить». В обработчике события нажатия на кнопку «Применить» происходит, в зависимости от выбора пункта меню, добавление новой записи или обновление записи в таблице Продуктов (**Products**). Код создания Диалогового Окна **android.app.AlertDialog**, вместе с назначением ему обработчика события нажатия на кнопку «Применить», приведен в Листинге 4.28. Чтобы отметить код, непосредственно относящийся к использованию

Адаптера Данных **android.widget.SimpleAdapter**, в Листинге 4.28 он выделен жирным шрифтом.

**Листинг 4.28.** Создание Диалогового окна **android.app.AlertDialog** и назначение ему обработчика события нажатия на кнопку «При-менить», в котором выполняется внесение изменений в таблицу Products Базы Данных и последующее обновление Адаптера Данных **android.widget.SimpleCursorAdapter**

```
@Override  
protected void onCreate(Bundle savedInstanceState)  
{  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
  
    ...  
  
    /*  
     * Создание объекта Диалогового окна  
     * android.app.AlertDialog для добавления /  
     * редактирования данных Продукта  
     * -----  
     */  
    //--- AlertDialog.Builder – объект построитель  
    //--- Диалоговых окон -----  
    AlertDialog.Builder builder =  
        new AlertDialog.Builder(this,  
            AlertDialog.THEME_HOLO_LIGHT);  
  
    //--- Обработчик кнопки "Отменить" -----  
    builder.setNegativeButton("Отменить",  
        new DialogInterface.OnClickListener()  
    {  
        @Override  
        public void onClick(DialogInterface dialog,  
            int which)  
        {
```

```
        Toast.makeText(MainActivity.this,
                  "Отменено", Toast.LENGTH_SHORT).show();
    }
});
```

//-- Обработчик кнопки "Применить" -----

```
builder.setPositiveButton("Применить",
                           new DialogInterface.OnClickListener()
{
    @Override
    public void onClick(DialogInterface dialog,
                        int which)
    {

//-- Чтение данных Продукта -----
        try
        {
            String productNameStr = ((EditText)
                ((AlertDialog) dialog).
                findViewById(R.id.edtName)).
                getText().toString();
            String productPriceStr = ((EditText)
                ((AlertDialog) dialog).
                findViewById(R.id.edtPrice)).
                getText().toString();
            String productWeightStr = ((EditText)
                ((AlertDialog) dialog).
                findViewById(R.id.edtWeight)).
                getText().toString();

            if (productNameStr.isEmpty() ||
                productPriceStr.isEmpty() ||
                productWeightStr.isEmpty())
            {
                throw new Exception(
                    "Заполнены не все поля!");
            }
        }
    }
});
```

```
        double productPrice = Double.  
            parseDouble(productPriceStr);  
        int productWeight = Integer.  
            parseInt(productWeightStr);  
  
        //--- Создание объекта ContentValues для внесения  
        //--- в таблицу БД -----  
        ContentValues row = new ContentValues();  
        row.put(MySQLiteOpenHelper.colProductName,  
                productNameStr);  
        row.put(MySQLiteOpenHelper.  
                colProductPrice, productPrice);  
        row.put(MySQLiteOpenHelper.  
                colProductWeight, productWeight);  
  
        //--- Определение, что необходимо сделать -  
        //--- добавить строку или заменить-----  
        String strHiddenId = ((TextView)  
            (AlertDialog) dialog).  
            findViewById(R.id.tvHidden)).  
            getText().toString();  
        if (strHiddenId.isEmpty())  
        {  
        //--- Добавление продукта -----  
            long rowID = db.insert(  
                MySQLiteOpenHelper.  
                tblNameProducts, null, row);  
  
            if (rowID == -1)  
                throw new Exception(  
                    "Строка не добавлена в таблицу  
(row ID = -1)");  
  
            Toast.makeText(MainActivity.this,  
                "Продукт успешно добавлен",  
                Toast.LENGTH_SHORT).show();
```

```
//-- Обновление Курсора -----
    MainActivity.this.adapter.swapCursor(
        db.query(MySQLiteOpenHelper.
            tblNameProducts, null, null, null,
            null, null, MySQLiteOpenHelper.
            colProductName)).close();
    }
else
{
//-- Обновление Продукта -----
    int cnt = db.update(MySQLiteOpenHelper.
        tblNameProducts, row, "_id=?",
        new String[] { strHiddenId });

    Toast.makeText(MainActivity.this,
        "Обновлено строк: " + cnt,
        Toast.LENGTH_SHORT).show();

    if (cnt > 0)
    {
//-- Обновление Курсора -----
        MainActivity.this.adapter.
            swapCursor(db.query(
                MySQLiteOpenHelper.
                    tblNameProducts,
                null, null, null, null, null,
                MySQLiteOpenHelper.
                    colProductName)).close();
    }
}
catch (Exception e)
{
    Toast.makeText(MainActivity.this,
        "Данные введены неправильно: " +
        e.getMessage(),
        Toast.LENGTH_SHORT).show();
}
```

```

        Log.d(TAG, e.getMessage());
    }
}
});

//-- Диалоговое окно для ввода данных -----
this.dialog = builder.create();
this.dialog.setView(this.getLayoutInflater().
    inflate(R.layout.dialog_add_item, null, true));
}
}

```

И последнее, в Листинге 4.29 приведен код метода **onOptionsItemSelected()**, который обрабатывает события выбора пунктов меню приложения. В этом методе отображается Диалоговое Окно из Листинга 4.28 для добавления или редактирования Продукта, а также осуществляется удаление выбранного Продукта из таблицы **Products** Базы Данных. Интересующий нас код в Листинге 4.29 выделен жирным шрифтом.

**Листинг 4.29.** Код метода **onOptionsItemSelected()**, обрабатывающего события выбора пунктов меню Приложения

```

@Override
public boolean onOptionsItemSelected(MenuItem item)
{
    //-- Получение доступа к БД -----
    final SQLiteDatabase db = this.dbHelper.
        getWritableDatabase();

    //-- Обработка выбранного пункта меню -----
    int id = item.getItemId();
    switch (id)
    {
    /*
     *  Добавление записи в БД

```

```
* -----
*/
        case R.id.action_add:
    {
//-- Диалоговое окно "Добавить новый продукт" ---
        this.dialog.setTitle(
            "Добавить новый Продукт");

//-- Показываем Диалоговое окно -----
        this.dialog.show();

//-- Очистка текстовых полей Диалогового окна ---
        if (this.dialog.findViewById(R.
            id.edtName) != null)
        {
            ((EditText) this.dialog.findViewById(
                R.id.edtName )).setText("");
            ((EditText) this.dialog.findViewById(
                R.id.edtPrice )).setText("");
            ((EditText) this.dialog.findViewById(
                R.id.edtWeight)).setText("");
            ((TextView) this.dialog.findViewById(
                R.id.tvHidden )).setText("");
        }
        break;

/*
 * Удаление записи из БД
 */
        case R.id.action_del:
//-- Находим текущий выбранный продукт -----
        if (this.curItem == -1)
        {
            Toast.makeText(this, "Продукт не выбран",
                Toast.LENGTH_SHORT).show();
        }
    }
}
```

```
        break;
    }

    Cursor C = this.adapter.getCursor();
    C.moveToPosition(this.curItem);
    int indexId = C.getColumnIndex(
        SQLiteOpenHelper.colId);
    int rowAffected = db.delete(
        SQLiteOpenHelper.tblNameProducts,
        "_id=?", new String[]{ String.
            valueOf(C.getInt(indexId)) });

    Toast.makeText(this, "Удалено строк: " +
        rowAffected, Toast.LENGTH_SHORT).show();

    this.curItem = -1;

    //-- Обновление Курсора -----
    this.adapter.swapCursor(db.query(
        SQLiteOpenHelper.tblNameProducts,
        null, null, null, null, null,
        SQLiteOpenHelper.colProductName)).
        close();
    break;

/*
 *   Обновление записи из БД
 * -----
 */
    case R.id.action_edt:
    //-- Диалоговое окно "Редактировать Продукт" ---
        this.dialog.setTitle("Редактировать
            Продукт");

    //-- Находим текущий выбранный продукт -----
        if (this.curItem == -1)
    {
```

```
        Toast.makeText(this,
                      "Продукт не выбран",
                      Toast.LENGTH_SHORT).show();
        break;
    }

//--- Извлекаем данные о текущем редактируемом
//--- Продукте -----
    Cursor C1 = this.adapter.getCursor();
    C1.moveToPosition(this.curItem);

    int indexID = C1.getColumnIndex(
        MySQLiteOpenHelper.colId);
    int indexName = C1.getColumnIndex(
        MySQLiteOpenHelper.colProductName);
    int indexPrice = C1.getColumnIndex(
        MySQLiteOpenHelper.colProductPrice);
    int indexWeight = C1.getColumnIndex(
        MySQLiteOpenHelper.colProductWeight);

    String strID = C1.getString(indexID);
    String strName = C1.getString(indexName);
    String strPrice = C1.getString(indexPrice);
    String strWeight = C1.getString(indexWeight);

//--- Показываем Диалоговое окно -----
    this.dialog.show();

//--- Инициализация текстовых полей Диалогового окна
    if (this.dialog.findViewById(
        R.id.edtName) != null)
    {
        ((EditText) this.dialog.findViewById(
            R.id.edtName)).setText(strName);
        ((EditText) this.dialog.findViewById(
            R.id.edtPrice)).setText(strPrice);
        ((EditText) this.dialog.findViewById(
            R.id.edtWeight)).setText(strWeight);
```

```
        ((TextView) this.dialog.findViewById( R.id.tvHidden )).setText(strID);
    }
    break;
}
return super.onOptionsItemSelected(item);
}
```

Как видно из Листинга 4.29, в коде обработки события выбора пункта меню «Добавить запись в Таблицу» по сравнению с кодом из Листинга 4.21 никаких изменений нет. Изменения для обработки пункта меню «Удалить запись из Таблицы» касаются только обновления Курсора, который назначается Адаптеру Данных с помощью метода **swapCursor()** (в Листинге 4.29 выделено жирным шрифтом). Причем предыдущий Курсор, ссылку на который возвращает метод **swapCursor()**, закрывается с помощью вызова метода **close()** и это правильно. И изменения для обработки пункта меню «Редактировать запись» касаются только извлечения данных редактируемой записи из Курсора и размещение этих данных в текстовых полях **android.widget.EditText** Диалогового Окна (в Листинге 4.29 выделено жирным шрифтом).

Подведем итог. Использование Адаптера Данных **android.widget.CursorAdapter** (в нашем случае **android.widget.SimpleCursorAdapter**) несколько упрощает написание приложения, которое взаимодействует с Базой Данных. Однако автоматического изменения данных в Адаптере Данных после внесения изменений в Базу Данных не происходит (а вообще ожидалось). После внесения изменений в Базу Данных необходимо обновлять Курсор (**android.database.Cursor**), который назначен Адаптеру Данных

**android.widget.SimpleCursorAdapter** с помощью кода, приведенного в Листинге 4.30. Иначе внесенные в Базу Данных изменения не отобразятся в списке **android.widget.ListView**, с которым ассоциирован Адаптер Данных.

**Листинг 4.30.** Обновление Курсора **android.database.Cursor** в Адаптере Данных **android.widget.SimpleCursorAdapter** после внесения изменений в Базу Данных для того, чтобы изменения отобразились в списке **android.widget.ListView**

```
//-- Обновление Курсора -----
this.adapter.swapCursor(
    db.query(
        MySQLiteOpenHelper.tblNameProducts,
        null, null, null, null, null,
        MySQLiteOpenHelper.colProductName)).close();
```

Внешний вид работы примера изображен на Рис. 4.2 и 4.3. Весь код примера данного раздела находится в модуле «app8» проекта с файлами исходных кодов, которые прилагаются к этому уроку.

#### **4.7. Пример взаимодействия Android приложения с многотабличной БД и управления изменениями версий Базы Данных**

Предыдущие, рассматриваемые в этом уроке примеры приложений, взаимодействующих с Базой Данных SQLite, были примерами взаимодействия с Базой Данных, состоящей из одной таблицы. Однако, чаще всего Базы Данных состоят из нескольких взаимосвязанных между собой таблиц. Это вам должно быть известно еще по курсу «Теория Баз Данных». Если вы не можете вспомнить, о чем сейчас идет речь, — остановитесь на этом месте

и перечитайте уроки по «Теории Баз Данных» и только затем продолжите знакомство с этим разделом.

В этом разделе рассматривается пример Android приложения, взаимодействующего с Базой Данных, которая состоит из 2-х таблиц: таблицы Продуктов (**Products**) и таблицы Категорий продуктов (**Categories**). Т.е. в Базу Данных из предыдущего раздела добавляется одна таблица «Категории» (**Categories**) и в таблицу «Продукты» (**Products**) добавляется столбец, который является внешним ключом для связи с таблицей «Категории» (название столбца **id\_category**). Другими словами, у Продуктов добавляется Категория продукта (например «Напитки», «Батончики» и т.д.). И поскольку речь идет о модификации структуры Базы Данных, то, пользуясь такой возможностью, мы затронем и пример управления версией Базы Данных (внесение изменений в структуру Базы Данных в методах обратного вызова **onUpgrade()** и **onDowngrade()** объекта **android.database.sqlite.SQLiteOpenHelper**).

Для рассматриваемого примера создан модуль «app9» в проекте, который прилагается этому уроку.

Итак, для того чтобы четко представить себе структуру Базы Данных (таблицы **Categories** и **Products**), в Листинге 4.31 представлены SQL команды создания этих таблиц.

#### Листинг 4.31. Структура Базы Данных для рассматриваемого в данном разделе примера

```
CREATE TABLE Categories
(
    _id integer not null primary key autoincrement,
        -- Идентификатор
    name text
        -- Название категории
);
```

```
CREATE TABLE Products
(
    _id integer not null primary key autoincrement,
        -- Идентификатор
    name    text,
        -- Название продукта
    price   real,
        -- Стоимость продукта
    weight  int,
        -- Вес продукта
    id_category  int
        -- Идентификатор категории.
        -- Внешний ключ
);
```

Однако, как сообщалось ранее, База Данных (**MyDbThree**) не будет создаваться с помощью команд из Листинга 4.31. Первоначально в ней будет только таблица Продуктов (**Products**), и это будет версия 1.0 Базы Данных **MyDbThree**. Команда создания таблицы **Products** приведена в Листинге 4.1 (только название столбца первичного ключа не «**id**», как в Листинге 4.1, а «**\_id**»). Затем в рассматриваемом примере произойдет переход Базы Данных на версию 2.0, чтобы продемонстрировать работу метода обратного вызова **onUpgrade()** класса **MySQLiteOpenHelper** (производный от **android.database.sqlite.SQLiteOpenHelper**). Сразу сообщим, что пример данного раздела создан с возможностью вернуть Базу Данных **MyDbThree** обратно — с версии 2.0 на версию 1.0, чтобы продемонстрировать работу метода обратного вызова **onDowngrade()** класса **MySQLiteOpenHelper**. Код класса **MySQLiteOpenHelper** для примера текущего раздела приведен в Листинге 4.32. Сообщим даже больше, чем ранее сказанное: в примере есть возможность многократно переходить с версии 1.0 Базы Данных **MyDbThree** на версию 2.0 и обратно. Для этого всего лишь необходимо

менять значение статического поля **dbVersion** класса **MySQLiteOpenHelper** (см. Листинг 4.32).

**Листинг 4.32.** Код класса **MySQLiteOpenHelper**, демонстрирует переходы Базы Данных с версии 1.0 на версию 2.0 и обратно

```
/**  
 * Class MySQLiteOpenHelper  
 * -----  
 */  
class MySQLiteOpenHelper extends SQLiteOpenHelper  
{  
    //-- Class constants -----  
    /**  
     * Tag для Log.d  
     */  
    private final static String TAG = "===== MySQLite";  
  
    /**  
     * Название Базы Данных  
     */  
    private final static String dbName = "MyDbThree";  
  
    /**  
     * Версия Базы Данных  
     */  
    public final static int dbVersion = 1;  
  
    /**  
     * Название таблицы Продуктов (Products) в Базе Данных  
     */  
    public final static String tblNameProducts =  
        "Products";  
  
    /**  
     * Имя столбца "name" таблицы Продуктов (Products)  
     */  
    public final static String colProductName = "name";
```

```
/**  
 * Имя столбца "price" таблицы Продуктов (Products)  
 */  
public final static String colProductPrice = "price";  
  
/**  
 * Имя столбца "weight" таблицы Продуктов (Products)  
 */  
public final static String colProductWeight=  
    "weight";  
  
/**  
 * Имя столбца "_id" таблицы Продуктов (Products)  
 */  
public final static String colId = "_id";  
  
//--- Class methods -----  
public MySQLiteOpenHelper(Context context)  
{  
    super(context, MySQLiteOpenHelper.dbName,  
          null, MySQLiteOpenHelper.dbVersion);  
}  
  
/**  
 * Метод обратного вызова. Вызывается, когда  
 * База Данных создается впервые на устройстве.  
 * В этом методе необходимо создать таблицы Базы  
 * Данных и, при необходимости, заполнить их  
 * начальными значениями. @param db Объект  
 * android.database.sqlite.SQLiteDatabase,  
 * с помощью которого можно отсыпать SQL запросы  
 * к Базе Данных.  
 */  
@Override  
public void onCreate (SQLiteDatabase db)  
{  
    Log.d(TAG, "onCreate: " + db.getPath());
```

```
/*
 * Создание таблиц
 * -----
 */
//-- Создание таблицы Products -----
String query = "CREATE TABLE Products(" +
    "_id integer not null primary key
        autoincrement, " +
    "name text, " +
    "price real, " +
    "weight integer)";

db.execSQL(query);

//-- Заполнение таблицы несколькими продуктами --
ContentValues row = new ContentValues();
row.put("name", "Snickers");
row.put("price", 12.50);
row.put("weight", 45);
db.insert(MySQLiteOpenHelper.tblNameProducts,
    null, row);

row = new ContentValues();
row.put("name", "Mars");
row.put("price", 13.90);
row.put("weight", 50);
db.insert(MySQLiteOpenHelper.tblNameProducts,
    null, row);

row = new ContentValues();
row.put("name", "Bounty");
row.put("price", 15.30);
row.put("weight", 60);
db.insert(MySQLiteOpenHelper.tblNameProducts,
    null, row);
}
```

```
/**  
 * Метод обратного вызова вызывается, когда База  
 * Данных нуждается в обновлении в связи  
 * с изменением номера версии Базы Данных на  
 * более новую.  
 * В этом методе необходимо выполнить действия по  
 * изменению структуры Базы Данных:  
 * удалить таблицы, добавить таблицы и т.д.  
 * @param db Объект android.database.sqlite.  
 * SQLiteDatabase,  
 * с помощью которого можно отсыпать SQL запросы  
 * к Базе Данных.  
 * @param oldVersion Номер предыдущей версии БД.  
 * @param newVersion Номер текущей версии БД.  
 */  
  
@Override  
public void onUpgrade (SQLiteDatabase db,  
                      int oldVersion, int newVersion)  
{  
    Log.d(TAG, "onUpgrade: " + db.getPath() +  
           "; oldVersion: " + oldVersion +  
           "; newVersion: " + newVersion);  
  
    if (oldVersion == 1 && newVersion == 2)  
    {  
        /*  
         * Переход Базы Данных MySqlTwo с версии 1.0  
         * на версию 2.0  
         * -----  
         */  
        //--- Создание таблицы Категорий продуктов -----  
        String query =  
            "CREATE TABLE Categories" + "(" +  
            "_id integer not null primary key  
            autoincrement," + "name text" + ")";  
  
        db.execSQL(query);  
    }  
}
```

```
//-- Заполнение таблицы Категорий несколькими
//-- категориями -----
    ContentValues row = new ContentValues();
    row.put("name", "Кондитерские изделия");
    db.insert("Categories", null, row);

    row = new ContentValues();
    row.put("name", "Напитки");
    db.insert("Categories", null, row);

    row = new ContentValues();
    row.put("name", "Молочные изделия");
    db.insert("Categories", null, row);

//-- Модификация таблицы Products – добавление
//-- столбца id_category –
    query = "ALTER TABLE Products
        ADD COLUMN id_category integer";
    db.execSQL(query);

//-- Назначение продуктам значения для id_category
    row = new ContentValues();
    row.put("id_category", 1);
    db.update(MySQLiteOpenHelper.
        tblNameProducts, row, null, null);
}

}

/**
 * Метод обратного вызова, не обязательный для
 * переопределения. Вызывается, когда База Данных
 * нуждается в обновлении в связи с изменением
 * номера версии Базы Данных на более старую.
 * В этом методе необходимо выполнить действия
 * по изменению структуры Базы Данных:
 * удалить таблицы, добавить таблицы и т.д.
 * @param db Объект android.database.sqlite.
```

```
* SQLiteDatabase, с помощью которого можно
* отсыпать SQL запросы к Базе Данных.
* @param oldVersion Номер предыдущей версии БД.
* @param newVersion Номер текущей версии БД.
*/
@Override
public void onDowngrade (SQLiteDatabase db,
                        int oldVersion, int newVersion)
{
    Log.d(TAG, "onDowngrade: " + db.getPath() +
          "; oldVersion: " + oldVersion +
          "; newVersion: " + newVersion);

    if (oldVersion == 2 && newVersion == 1)
    {
/*
     * Возврат БД MySqlTwo с версии 2.0 на версию 1.0
     * -----
     */
    //--- Переименование таблицы Products в OldProducts
    String queryc = "ALTER TABLE Products
                     RENAME TO OldProducts";
    db.execSQL(query);

    //--- Создание таблицы Products без столбца
    //--- id_category -----
    query = "CREATE TABLE Products(" +
            "_id integer not null primary
key autoincrement, " +
            "name text, " +
            "price real, " +
            "weight integer)";

    db.execSQL(query);

    //--- Перенос строк из OldProducts в Products без
    //--- столбца id_category
```

```
Cursor C = db.query(
    "OldProducts",
    null, null, null, null, null, null);

if (C.moveToFirst())
{
//-- Получение индексов столбцов -----
    int indexId = C.getColumnIndex(
        SQLiteOpenHelper.colId);
    int indexName = C.getColumnIndex(
        SQLiteOpenHelper.colProductName);
    int indexPrice = C.getColumnIndex(
        SQLiteOpenHelper.colProductPrice);
    int indexWeight = C.getColumnIndex(
        SQLiteOpenHelper.colProductWeight);

//-- Переход по всем строкам в Cursor -----
    do
    {
        ContentValues row =
            new ContentValues();
        row.put(
            SQLiteOpenHelper.colId,
            C.getInt(indexId));
        row.put(
            SQLiteOpenHelper.colProductName,
            C.getString(indexName));
        row.put(
            SQLiteOpenHelper.colProductPrice,
            C.getDouble(indexPrice));
        row.put(
            SQLiteOpenHelper.colProductWeight,
            C.getInt(indexWeight));
        db.insert(SQLiteOpenHelper.
            tblNameProducts, null, row);
    }
}
```

```
        while (C.moveToNext());
        C.close();
    }
//-- Удаление таблицы Categories -----
    query = "DROP TABLE Categories";
    db.execSQL(query);
}
}
```

В Листинге 4.32 видно, что при изменении версии Базы Данных с меньшей на большую (в примере — с версии 1.0 на версию 2.0) вызывается метод обратного вызова `onUpgrade()` для объекта класса `MySQLOpenHelper`. В этом методе происходит создание таблицы `Categories` с помощью SQL команды, как показано в Листинге 4.31, и происходит добавление столбца `id_category` в таблицу `Products` с помощью команды `ALTER TABLE`:

```
ALTER TABLE Products ADD COLUMN id category integer;
```

И затем всем продуктам для столбца `id_category` присваивается значение 1 (категория «Кондитерские изделия») с помощью вызова метода `update()`, как показано в Листинге 4.33.

**Листинг 4.33.** Присвоение значения всем строкам таблицы Products для вновь добавленного столбца id\_category

```
//-- Назначение продуктам значения для id_category --
row = new ContentValues();
row.put("id_category",    1);
db.update(MySQLiteOpenHelper.tblNameProducts, row,
        null, null);
```

Если в примере произойдет изменение версии с большей на меньшую (с версии 2.0 на 1.0), то запустится метод обратного вызова **onDowngrade()** (см. Листинг 4.32). В этом методе вначале удаляется столбец **id\_category**, а затем удаляется таблица **Categories**. К сожалению, в СУБД SQLite нет команды удаления столбцов из таблицы:

```
ALTER TABLE ИмяТаблицы DROP COLUMN имяСтолбца;
```

Хотя подобная команда SQL существует для других типов СУБД. Поэтому удаление столбца **id\_category** происходит следующим путем:

- Вначале таблица **Products** переименовывается в таблицу **OldProducts** с помощью команды SQL:

```
ALTER TABLE Products RENAME TO OldProducts;
```

- Затем создается таблица **Products** (см. Листинг 4.31), но уже без столбца **id\_category**.
- Делается выборка всех строк из таблицы **OldProducts** в объект **android.database.sqlite.SQLiteDatabase** и затем каждая строка из этого Курсора вставляется в таблицу **Products**.

Когда будете запускать пример данного раздела (модуль «app9»), вначале запустите со значением статического поля **dbVersion** (класс **MySQLiteOpenHelper**) равным 1. Это необходимо для того, чтобы создалась База Данных **MyDbThree** версии 1.0 (только таблица **Products** без категорий продуктов). Затем поменяйте значение статического поля **dbVersion** на значение 2, что будет соответствовать

версии Базы Данных 2.0. В Листинге 4.34 показано, как поменять значение поля **dbVersion** на значение 2.

**Листинг 4.34.** Изменение версии Базы Данных MyDbThree из рассматриваемого примера

```
/**  
 * Class MySQLiteOpenHelper  
 * -----  
 */  
class MySQLiteOpenHelper extends SQLiteOpenHelper  
{  
//--- Class constants -----  
...  
  
/**  
 * Версия Базы Данных  
 */  
public final static int dbVersion = 2;  
  
...  
}
```

Для того чтобы вернуться обратно к версии Базы Данных 1.0, необходимо присвоить статическому полу **dbVersion** значение 1.

В классе главной Активности приложения в методе **onCreate()** создается Адаптер Данных ([android.widget.SimpleCursorAdapter](#)), который отображает содержимое таблицы Продуктов ([Products](#)). Причем для версии Базы Данных 1.0 каждый элемент списка Продуктов отображается с помощью набора виджетов из файла ресурсов / [res/layout/product\\_item.xml](#) (см. Листинг 4.14). Для версии Базы Данных 2.0 появилась необходимость отображать

для каждого Продукта Категорию, поэтому для набора виджетов представляющих каждый элемент такого списка создан файл ресурсов /res/layout/product\_item\_with\_category.xml. Содержимое этого файла ресурсов приведено в Листинге 4.35. Внешний вид списков Продуктов для разных версий Базы Данных изображен на Рис. 4.4.



**Рис. 4.4.** Внешний вид списков Продуктов для разных версий Базы Данных рассматриваемого примера

**Листинг 4.35.** Xml верстка набора виджетов для элементов списка Продуктов версии 2.0 Базы Данных MyDbThree

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
```

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="10pt"
    android:layout_gravity="center_horizontal"
    android:layout_margin="2dp"
    android:id="@+id/tvName"
    android:textColor="#0F5119" />

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="8pt"
    android:textStyle="italic"
    android:layout_gravity="center_horizontal"
    android:layout_margin="2dp"
    android:id="@+id/tvCategory"
    android:textColor="#5f0f63" />

<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="8pt"
        android:id="@+id/tvPrice"
        android:gravity="center_horizontal"
        android:layout_weight="1"
        android:textColor="#0F5119" />
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="8pt"
        android:id="@+id/tvWeight"
```

```

        android:gravity="center_horizontal"
        android:layout_weight="1"
        android:textColor="#0F5119" />
    </LinearLayout>
</LinearLayout>

```

Создание Адаптера Данных в зависимости от версии Базы Данных осуществляется в методе **onCreate()** класса Активности и приведено в Листинге 4.36.

**Листинг 4.36.** Создание объекта Адаптера Данных android.widget.SimpleCursorAdapter для отображения содержимого таблицы Products в списке android.widget.ListView

```

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

    //--- DBHelper -----
    this.dbHelper = new MySQLiteOpenHelper(this);

    //--- Получение ссылки на ListView (lvMain) -----
    ListView LV = (ListView) this.findViewById(R.
        id.lvMain);

    //--- Получение ссылки на объект android.database.
    //--- sqlite.SQLiteDatabase -----
    final SQLiteDatabase db = this.dbHelper.
        getWritableDatabase();

    //--- Устанавливаем значение версии Базы Данных
    //--- в текстовое поле tvVersion
    String strVersion = "Версия Базы Данных " +
        MySQLiteOpenHelper.dbVersion;

```

```
((TextView) this.findViewById(R.id.tvVersion)).  
    setText(strVersion);  
  
    //--- Отображение списка продуктов в зависимости  
    //--- от версии БД -----  
    if (MySQLiteOpenHelper.dbVersion == 2)  
    {  
    /*  
     * ВЕРСИЯ 2.0  
     * Считывание данных из БД и заполнение ими ListView  
     * -----  
     */  
    //--- Raw SQL многотабличный запрос -----  
    String query =  
        "SELECT Products._id AS _id,  
            Products.name AS pname, " +  
            "Products.price AS price, " +  
            "Products.weight AS weight,  
            Categories.name AS cname " +  
            "FROM Products, Categories " +  
            "WHERE Products.id_category =  
            Categories._id " +  
            "ORDER BY pname";  
  
    Cursor C = db.rawQuery(query, null);  
  
    /*  
     * SimpleCursorAdapter (Адаптер Данных) для ListView  
     * -----  
     */  
    this.adapter = new SimpleCursorAdapter(  
        this,  
        R.layout.product_item_with_category,  
        C,  
        new String[]  
        {  
            "pname",  
            "price",  
            "weight",  
            "cname"  
        },  
        new int[]  
        {  
            R.id.pname,  
            R.id.price,  
            R.id.weight,  
            R.id.cname  
        }  
    );  
    this.listView.setAdapter(this.adapter);  
    this.listView.setOnItemClickListener(this);  
}
```

```
        "cname",
        "price",
        "weight"
    },
    new int[]
    {
        R.id.tvName,
        R.id.tvCategory,
        R.id.tvPrice,
        R.id.tvWeight
    },
    0);

    LV.setAdapter(this.adapter);
}
else
if (MySQLiteOpenHelper.dbVersion == 1)
{
/*
 * ВЕРСИЯ 1.0
 * Считывание данных из БД и заполнение ими ListView
 * -----
 */
    Cursor C = db.query(
        MySQLiteOpenHelper.tblNameProducts,
        null, null, null, null, null,
        MySQLiteOpenHelper.colProductName);

/*
 * SimpleCursorAdapter (Адаптер данных) для ListView
 * -----
 */
    this.adapter = new SimpleCursorAdapter(
        this,
        R.layout.product_item,
        C,
        new String[]
```

```
        {
            MySQLiteOpenHelper.colProductName,
            MySQLiteOpenHelper.colProductPrice,
            MySQLiteOpenHelper.colProductWeight
        },
        new int[]
        {
            R.id.tvName,
            R.id.tvPrice,
            R.id.tvWeight
        },
        0);
LV.setAdapter(this.adapter);
}
}
```

Хотим обратить внимание на тот факт, что в версии 2.0 База Данных **MyDbThree** становится многотабличной Базой Данных, так как в ней появляются взаимосвязанные таблицы (**Products** и **Categories**). Для выборки информации о Продуктах вместе с названием Категории необходим многотабличный запрос. В классе **android.database.sqlite.SQLiteDatabase** для исполнения многотабличных запросов предназначен метод **rawQuery()**. Этот метод принимает SQL запрос в «чистом виде» (перевод слова **raw** — «сырой»). Передавая методу **rawQuery()** многотабличный запрос, метод в результате его исполнения вернет ссылку на объект **android.database.sqlite.SQLiteDatabase**, принципы работы с которым уже рассмотрен нами в предыдущих разделах этого урока. В Листинге 4.36 вызов метода **rawQuery()** для многотабличного запроса выделен жирным шрифтом. В дальнейшем, результат исполнения многотабличного запроса обрабатывается аналогично

результату исполнения обычного однотабличного запроса. Обратите внимание, что столбцам для результирующей таблицы даются псевдонимы (**pname**, **cname** и т.д.), чтобы не иметь проблем с выборкой значений для этих столбцов из Адаптера Данных.

Внешний вид работы примера из Листинга 4.36 изображен на Рис. 4.4.

Весь исходный код примера данного раздела находится в модуле «app9» проекта с исходными кодами, который прилагается к этому уроку.

# 5. Домашнее задание

1. Доработайте пример из Листингов 1.26–1.29 (модуль «app4») таким образом, чтобы для отображаемого в Фрагменте **android.app.ListFragment** списке «Контактов Электронных Адресов» появилась возможность добавлять, удалять и редактировать данные списка (без сохранения в файле или в Базе Данных). Для ввода пользовательских данных необходимо использовать Фрагмент **android.app.DialogFragment**.
2. «Активность поиска изображений». Создайте Активность, с помощью которой можно выполнять навигацию по каталогам Внешнего носителя устройства. В каталогах Активность отображает только файлы изображений в миниатюрном виде. Остальные типы файлов в Активности не отображаются. Активность позволяет выбрать пользователю необходимый файл. При этом сама Активность закрывается и возвращает в вызывающую Активность путь к выбранному файлу. Настройте intent фильтры для этой «Активности поиска изображений». Создайте другую Активность, которая бы вызывала «Активность поиска изображений» и, после ее закрытия, отображала бы в виджете **android.widget.ImageView** выбранный пользователем файл изображения с Внешнего носителя.
3. Приложение «База Данных Фильмов». Создайте приложение, которое будет работать с многотабличной Базой Данных Фильмов. В Базе Данных есть две

таблицы: Фильмы (**Films**) и Жанры (**Genres**). У Фильма есть следующие поля: «Название», «Год выпуска», «Жанр» (идентификатор жанра), «Постер» (строковый столбец, содержащий путь к файлу изображению). Приложение предоставляет пользователю добавлять, удалять редактировать как фильмы, так и жанры. Для назначения фильму Постера используйте «Активность поиска изображений» из предыдущего задания. Выбранный файл с изображением для Постера копируется на Внутренний носитель (в песочницу приложения) и в Базу Данных заносится только путь к файлу. В списке Фильмов **android.widget.ListView** (или **android.app.ListFragment** — на ваш выбор) для каждого Фильма, рядом с Названием, Годом выпуска, Жанром, отображается и Постер фильма в виджете **android.widget.ImageView**.





## Урок №4

# Детальный обзор виджетов Android

© Бояршинов Юрий

© Компьютерная Академия «Шаг»

[www.itstep.org](http://www.itstep.org)

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объеме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объем и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.