Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

РАЗРАБОТКА МОБИЛЬНЫХ ПРИЛОЖЕНИЙ

*Учебно-методическое пособие
для студентов специальностей 1-40 05 01 03 «Информационные системы и технологии (издательско-полиграфический комплекс)», 1-98 01 03 «Программное обеспечение информационной безопасности мобильных систем»*

Минск 2020

УДК 004.415(0.034.2)
ББК 32.972.11я73
Р17

Рассмотрено и рекомендовано редакционно-издательским советом Белорусского государственного технологического университета.

С о с т а в и т е л ь
Н. В. Пацей

Р е ц е н з е н т ы :
директор ООО «Ведекстим» *И. Ф. Саганович*;
кафедра программного обеспечения информационных технологий УО «Белорусский государственный университет информатики и радиоэлектроники»

Разработка мобильных приложений : электронное учеб.-
Р17 метод. пособие для студентов специальностей 1-40 05 01 03 «Информационные системы и технологии (издательско-полиграфический комплекс)», 1-98 01 03 «Программное обеспечение информационной безопасности мобильных систем» / сост. Н. В. Пацей. – Минск : БГТУ, 2020. – 265 с.

В данном учебно-методическом пособии представлены базовые темы для начала разработки приложений под мобильные устройства на платформе Android. Рассматриваются принципы создания пользовательского интерфейса, работы с данными, файловой системой, активностями, интентами, фрагментами, навигационными элементами и анимациями. Описаны базовые принципы построения архитектуры приложений.

УДК 004.415(0.034.2)
ББК 32.972.11я73

ВВЕДЕНИЕ

Android – операционная система для смартфонов, планшетов, электронных книг, цифровых проигрывателей, наручных часов, фитнес-браслетов, игровых приставок, ноутбуков, очков Google Glass, телевизоров и других устройств [1]. Она основана на ядре Linux и собственной реализации виртуальной машины Java от Google. Изначально разрабатывалась компанией Android Inc. (теперь Google).

Каждая версия системы начиная с версии 1.5 получает собственное кодовое имя на тему сладостей. Кодовые имена присваиваются в порядке латинского алфавита (рисунок) [1]. На настоящий момент выпущено 15 версий системы. Исходя из статистики на май 2019 г. [1].

Как показано на рисунке, самой распространенной версией остается Android 6.0, она установлена на 16,9% всех устройств. Следом за ней расположилась система Android 8.1 с 15,4%.

Версия	Название	Год	Доля
2.3	*Gingerbread*	2010	0,3 %
4.0	*Ice Cream Sandwich*	2011	0,3 %
4.1	*Jelly Bean*	2012	1,2 %
4.2		2012	1,5 %
4.3		2013	0,5 %
4.4	*KitKat*	2013	6,9 %
5.0	*Lollipop*	2014	3 %
5.1		2015	11,5 %
6.0	*Marshmallow*	2015	16,9 %
7.0	*Nougat*	2016	11,4 %
7.1		2016	7,8 %
8.0	*Oreo*	2017	12,9 %
8.1		2017	15,4 %
9.0	*Pie*	2018	10,4 %
10.0	*Q*	2019	< 0,1 %

Версии операционной системы Android

Для разработки приложений под операционную систему Android нужно скачать и установить SDK. Сейчас существует несколько сред разработки:

- NetBeans;
- Eclipse;
- Intellij IDEA;
- Android Studio.

Android Studio ориентирована именно под ОС Android, а также не требует установки дополнительных плагинов. Примеры выполнения заданий в данном пособии будут рассматриваться на Android Studio.

Рассмотрим языки разработки нативных приложений.

Java – официальный язык программирования, поддерживаемый средой разработки Android Studio. На Java ссылается большинство официальной документации Google.

Kotlin – язык был официально представлен в мае 2017 г. на Google I/O и позиционируется Google как второй официальный язык программирования под Android после Java. Kotlin совместим с Java и не вызывает снижения производительности и увеличения размера файлов. Отличие от Java в том, что он требует меньше служебного кода, поэтому более легкий для чтения.

Более низкоуровневые языки поддерживаются Android Studio с использованием Java NDK (Native Development Kit). Это позволяет писать нативные приложения, что может пригодиться для создания игр или других ресурсоемких программ.

Примеры, приведенные в данном учебно-методическом пособии, написаны на языке Java.

1. СОЗДАНИЕ ПРИЛОЖЕНИЙ НА ПЛАТФОРМЕ ANDROID

1.1. Создание первого Android-приложения

Для создания нового проекта в Android Studio (рис. 1.1) необходимо выбрать шабон приложения.

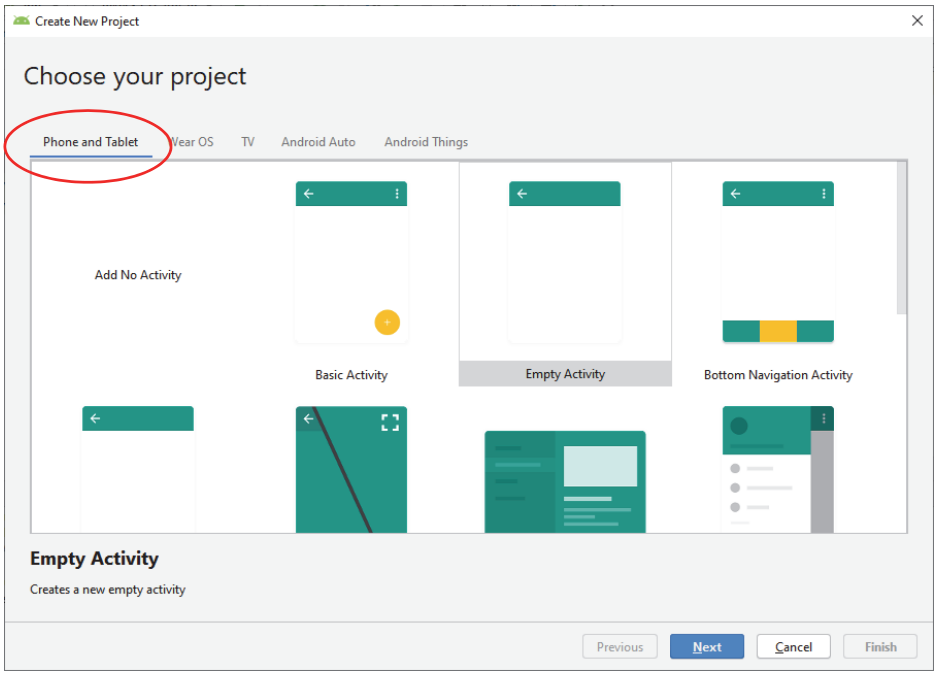


Рис. 1.1. Окно определения шаблона

Затем при задании параметра *Package name* (рис. 1.2) необходимо использовать обратное доменное имя, например *by.bstu.fit.фио*.

Следующий шаг – установка версии API (рис. 1.2). При выборе более нового API предоставляется небольшой процент поддерживаемых устройств.

1.1.1. Компоненты приложения

Компоненты приложения являются блоками, из которых состоит приложение. Каждый компонент представляет собой отдельную точку,

через которую система может войти в приложение. Не все компоненты являются точками входа для пользователя. Однако каждый компонент – это самостоятельная структурная единица, которая определяет работу приложения в целом. Компоненты приложения можно отнести к одному из пяти типов.

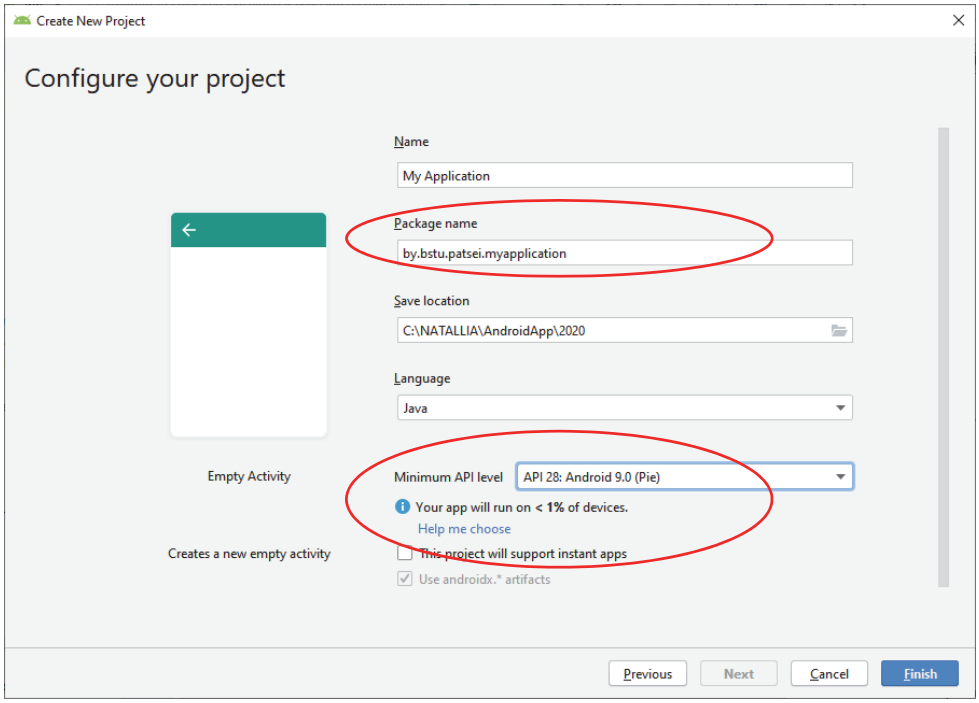


Рис. 1.2. Окно создания проекта Android-приложения

Операция (*Activity*, активность) представляет собой один экран с пользовательским интерфейсом. Например, в приложении для работы с электронной почтой одна активность может служить для отображения списка новых сообщений, другая – для составления сообщения, а третья – для чтения сообщений. Несмотря на то что активности совместно формируют взаимодействие пользователя с приложением, каждая из них не зависит от других. Любые из активностей могут быть запущены другим приложением. Например, приложение для камеры может запустить операцию в приложении по работе с электронной почтой, которая составляет новое сообщение, чтобы пользователь мог отослать фотографию. Операция относится к подклассу *Activity* [2].

Служба (*Service*, сервис) представляет собой компонент, который работает в фоновом режиме и выполняет длительные операции, связанные с работой удаленных процессов [2]. Служба не имеет пользовательского интерфейса. Например, она может воспроизводить музыку в фоновом режиме, пока пользователь работает в другом приложении,

или может получать данные по сети, не блокируя взаимодействие пользователя с активностью. Служба может быть запущена другим компонентом, который затем будет взаимодействовать с ней. Служба относится к подклассу *Service*.

Поставщик контента (*Content provider*) управляет общим набором данных приложения. Данные можно хранить в файловой системе, базе данных SQLite, в интернете или любом другом месте хранения, к которому у приложения есть доступ. Посредством поставщика контента другие приложения могут запрашивать или изменять данные (если поставщик контента позволяет делать это). Например, в системе Android есть поставщик контента, который управляет информацией контактов пользователя. Любое приложение, получившее соответствующие разрешения, может запросить часть этого поставщика контента для чтения и записи. Он относится к подклассу *ContentProvider* и должен реализовывать стандартный набор API-интерфейсов [2].

Приемник широковещательных сообщений (*Broadcast receiver*) представляет собой компонент, который реагирует на объявления, распространяемые по всей системе. Многие из этих объявлений рассылает система, например уведомления о том, что экран выключился, аккумулятор разряжен или был сделан фотоснимок. Объявления также могут рассылаться приложениями. Например, можно сообщить другим приложениям о том, что какие-то данные были загружены на устройство и теперь готовы для использования. Несмотря на то что приемники широковещательных сообщений не имеют пользовательского интерфейса, они могут создавать уведомления в строке состояния, чтобы предупредить пользователя о событии «рассылка объявления» [2]. Чаще всего они являются просто шлюзом для других компонентов и предназначены для выполнения минимального объема работы. Приемник широковещательных сообщений относится к подклассу *BroadcastReceiver*.

Уникальной особенностью системы Android является то, что любое приложение может запустить компонент другого приложения. Для пользователя это будет выглядеть как одно приложение. Когда система запускает компонент, она запускает процесс для этого приложения (если он еще не был запущен) и создает экземпляры классов, которые требуются этому компоненту. Поэтому в отличие от приложений для большинства других систем в приложениях для Android отсутствует единая точка входа (в них нет функции *main*()).

Каждое приложение выполняется системой в отдельном процессе с такими правами доступа к файлам, которые ограничивают доступ другим приложениям. Поэтому одно приложение не может напрямую

вызвать компонент из другого приложения. Но это может сделать система Android. Для того чтобы вызвать компонент в другом приложении, необходимо сообщить системе о своем **намерении** (*Intent*). После чего система активирует этот компонент.

1.1.2. Структура проекта

Вернемся к созданному проекту (рис. 1.3). В его структуре имеется единственный модуль – модуль *app*. Код располагается внутри этого модуля.

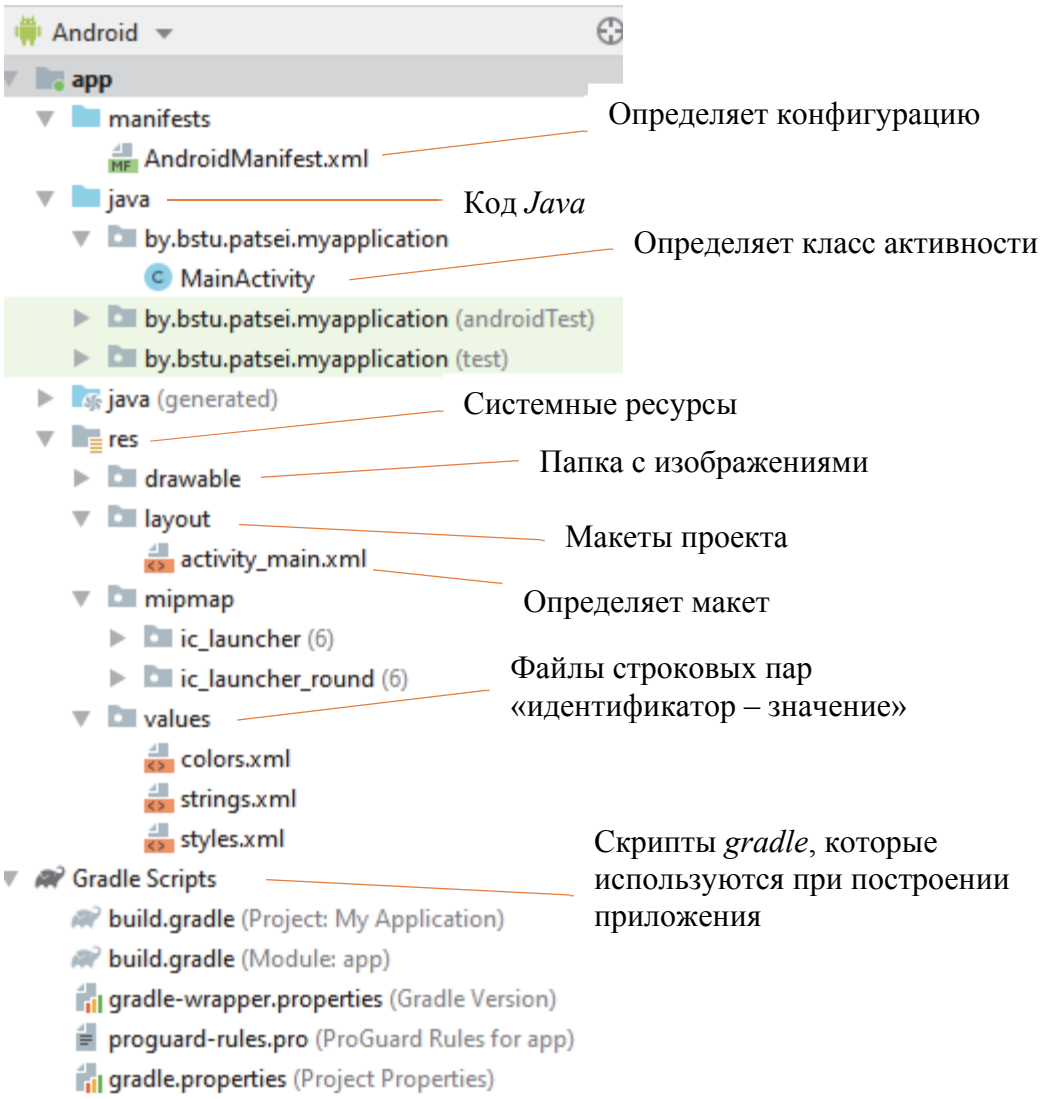


Рис. 1.3. Структура проекта Android-приложения

Все модули в проекте описываются файлом *setting.gradle*. По умолчанию он имеет следующее содержимое:

```
include ':app'
```

Каждый модуль имеет свой файл *build.gradle*, который определяет конфигурацию построения проекта. На начальном этапе данные файлы не столь важны, достаточно лишь понимать, для чего они нужны.

В *app* есть несколько папок и файлов.

Каталоги *androidTest* и *test* предназначены для хранения файлов тестов приложения, а каталог *java* – для хранения исходного кода. Класс *MainActivity* имеет следующую структуру:

```
package by.bstu.patsei.myapplication;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

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

Файл *AndroidManifest.xml* представляет собой файл манифеста, который описывает фундаментальные характеристики приложения, его конфигурацию и определяет каждый из компонентов данного приложения.

Для запуска компонента системе Android необходимо знать, что компонент существует. Для этого она читает файл *AndroidManifest.xml* приложения, который должен находиться в его корневой папке. Ниже приведен пример файла манифеста:

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

  <application
      android:allowBackup="true"
      android:icon="@mipmap/ic_launcher"
      android:label="@string/app_name"
      android:roundIcon="@mipmap/ic_launcher_round"
      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>
  </application>

</manifest>
```

Здесь *package* объявляет имя Java-пакета данного приложения, которое служит уникальным идентификатором приложения. Атрибут *android:name* в элементе <*activity*> указывает полное имя класса подкласса *Activity*, а атрибут *android:label* указывает строку, которую необходимо использовать в качестве метки активности, отображаемой для пользователя. Компоненты приложения описываются через атрибуты *activity*, *service*, *receiver* и *provider*. Эти объявления позволяют операционной системе узнать, чем компоненты являются и при каких условиях они могут быть запущены. Все компоненты приложения необходимо объявлять следующим образом:

— элементы <*activity*> – для операций;

— элементы <*service*> – для служб;

— элементы <*receiver*> – для приемников широковещательных сообщений;

— элементы <*provider*> – для поставщиков контента.

Системе не видны активности, службы и поставщики контента, которые имеются в исходном коде, но не объявлены в манифесте, поэтому они не могут быть запущены.

Манифест помимо объявления компонентов приложения служит и для других целей. Например, указание всех полномочий пользователя, которые требуются приложению (разрешения на доступ в интернет или на чтение контактов пользователя), для объявления минимального уровня API, требуемого приложению с учетом того, какие API-интерфейсы оно использует, для объявления аппаратных и программных функций, которые нужны приложению или используются им, например функции камеры, службы Bluetooth или сенсорного экрана, для указания библиотек API, с которыми необходимо связать приложение, например библиотеки Google Maps и др.

Папка *res* содержит каталоги с ресурсами (см. рис. 1.3):

— папка *drawable* предназначена для хранения изображений, используемых в приложении;

— папка *layout* предназначена для хранения файлов, определяющих графический интерфейс. По умолчанию здесь есть файл *activity_main.xml*, который определяет интерфейс для единственной в проекте активности – *MainActivity*;

— папки *mipmap*-xxxx содержат файлы изображений, которые предназначены для создания иконки приложения при различных разрешениях экрана. Соответственно для каждого вида разрешения здесь имеется свой каталог;

— папка *values* хранит различные XML-файлы, содержащие коллекции ресурсов.

Используя ресурсы приложения, можно без труда изменять его характеристики, не меняя код, и, кроме того, путем предоставления наборов альтернативных ресурсов можно оптимизировать приложение для работы с различными конфигурациями устройств (например, для различных языков или размеров экрана).

Для каждого ресурса, включаемого в проект Android, инструменты SDK задают уникальный целочисленный идентификатор, который может использоваться, чтобы сослаться на ресурс из кода приложения или из других ресурсов, определенных в XML. Например, если в приложении имеется файл изображения с именем *logo.png* (сохраненный в папке *res/drawable/*), инструменты SDK сформируют идентификатор ресурса под именем *R.drawable.logo*, с помощью которого на изображение можно будет ссылаться и вставлять его в пользовательский интерфейс.

Один из наиболее важных аспектов предоставления ресурсов отдельно от исходного кода заключается в возможности использовать альтернативные ресурсы для различных конфигураций устройств. Например, после определения строк пользовательского интерфейса в XML, можно перевести их на другие языки и сохранить эти переводы в отдельных файлах. Затем по квалификатору языка, добавленному к имени каталога ресурса, и выбранному пользователем языку система Android применит к вашему пользовательскому интерфейсу строки на соответствующем языке.

Android поддерживает разные квалификаторы для ресурсов. Квалификатор представляет собой короткую строку, которая включается в имена каталогов ресурсов с целью определения конфигурации устройства, для которой эти ресурсы следует использовать. Например, когда экран устройства имеет книжную ориентацию (расположен вертикально), кнопки в макете можно также размещать по вертикали, а когда экран развернут горизонтально (альбомная ориентация), кнопки следует размещать по горизонтали. Чтобы при изменении ориентации экрана изменялся макет, можно определить два разных макета и применить соответствующий квалификатор к имени каталога каждого макета. После этого система будет автоматически применять соответствующий макет в зависимости от ориентации устройства.

1.1.3. Разработка интерфейса приложения

В студии должен быть открыт файл *activity_main.xml*, который содержит определение графического интерфейса приложения.

Если файл открыт в режиме дизайнера, а в центре отображается дизайн приложения, то надо переключить вид файла в текстовый. Для переключения режима из текстового в графический и обратно внизу есть две кнопки *Design* и *Text* (рис. 1.4).

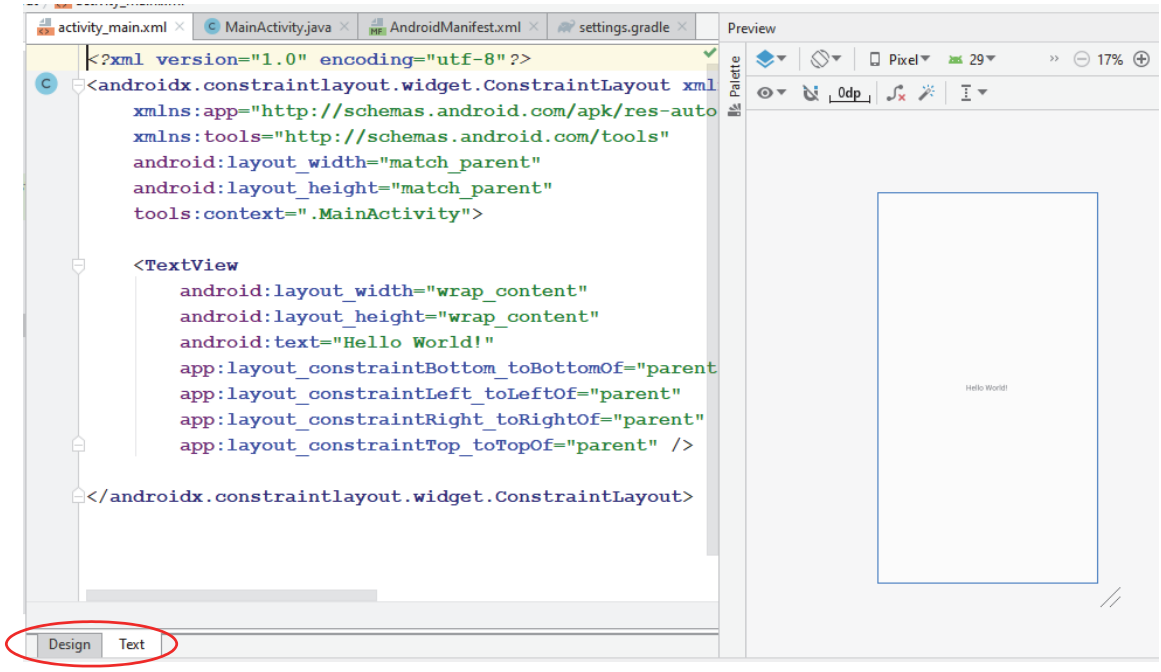


Рис. 1.4. Режимы редактирования Activity

Изменим код приложения так, чтобы оно выводило на экран строку «Hello World!». В файле *activity_main.xml* определение элемента *TextView* отвечает за вывод текстовой информации на экран. Выводимый текст задается с помощью атрибута *android:text*.

После сохранения файла можно переключиться в графический режим (см. рис. 1.4), где можно устанавливать режимы *Design* – в нем *view*-компоненты выглядят так, как обычно; *Blueprint* – отображаются только контуры *view*-компонентов; *Design + Blueprint* – два экрана одновременно. Кнопки на панели (рис. 1.5) позволяют переключать режимы *Design, Blueprint, Design + Blueprint*.

Окно *Палитра* (*Palette*) представляет собой список всех *view*-компонентов, которые можно добавлять на экран: кнопки, поля ввода, чекбоксы, прогрессбары и т. д.

Окно *Дерево компонентов* (*Component Tree*) отображает иерархию *view*-компонентов. Например, на рис. 1.5 корневой элемент – это *ConstraintLayout,* а в него вложен *TextView*.

В окне *Свойства* (*Properties*) при работе с *view*-компонентом будут отображаться его свойства (рис. 1.6).

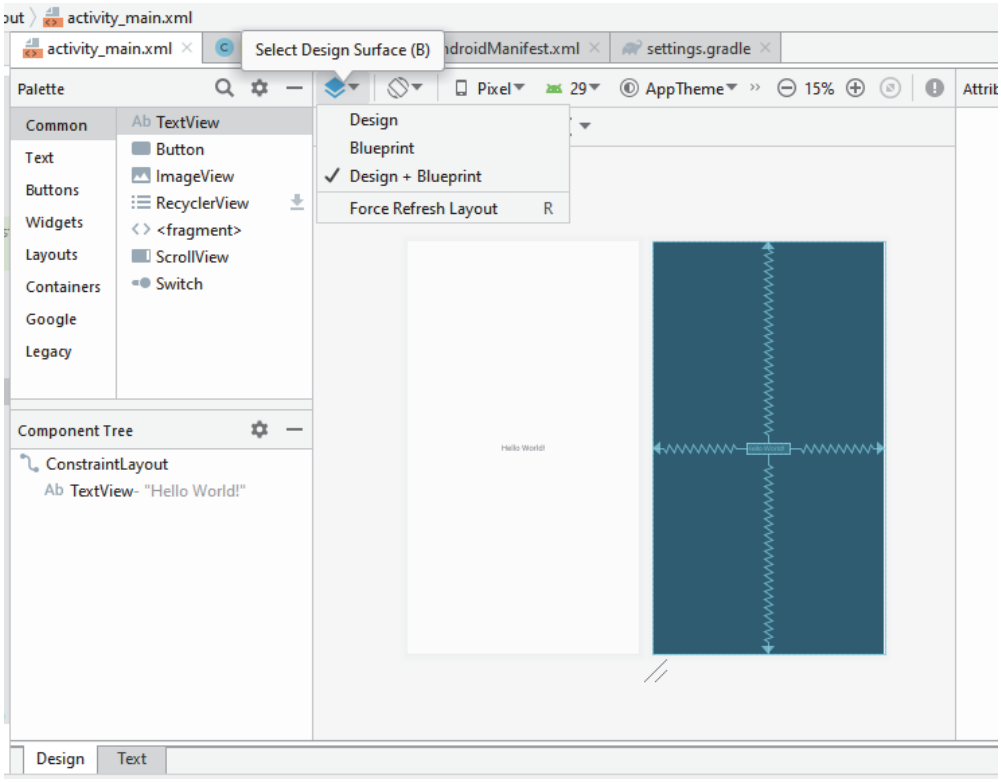


Рис. 1.5. Структура окон разработки и редактирования экрана приложения

Activity является классом, который по сути представляет отдельный экран (страницу) приложения или его визуальный интерфейс. Приложение может иметь одну операцию, а может и несколько. Каждая отдельная активность задает окно для отображения.

Рассмотрим код активности, которая генерируется автоматически в Android Studio (файл кода можно найти в проекте в папке *src/main/java*):

```
package by.bstu.patsei.myapplication;
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
public class MainActivity extends AppCompatActivity {

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

Класс *MainActivity* представляет обычный Java-класс, в начале которого идут определения пакета и импорта внешних пакетов. По умолчанию он содержит только один метод *onCreate*(), в котором фактически и создается весь интерфейс приложения.

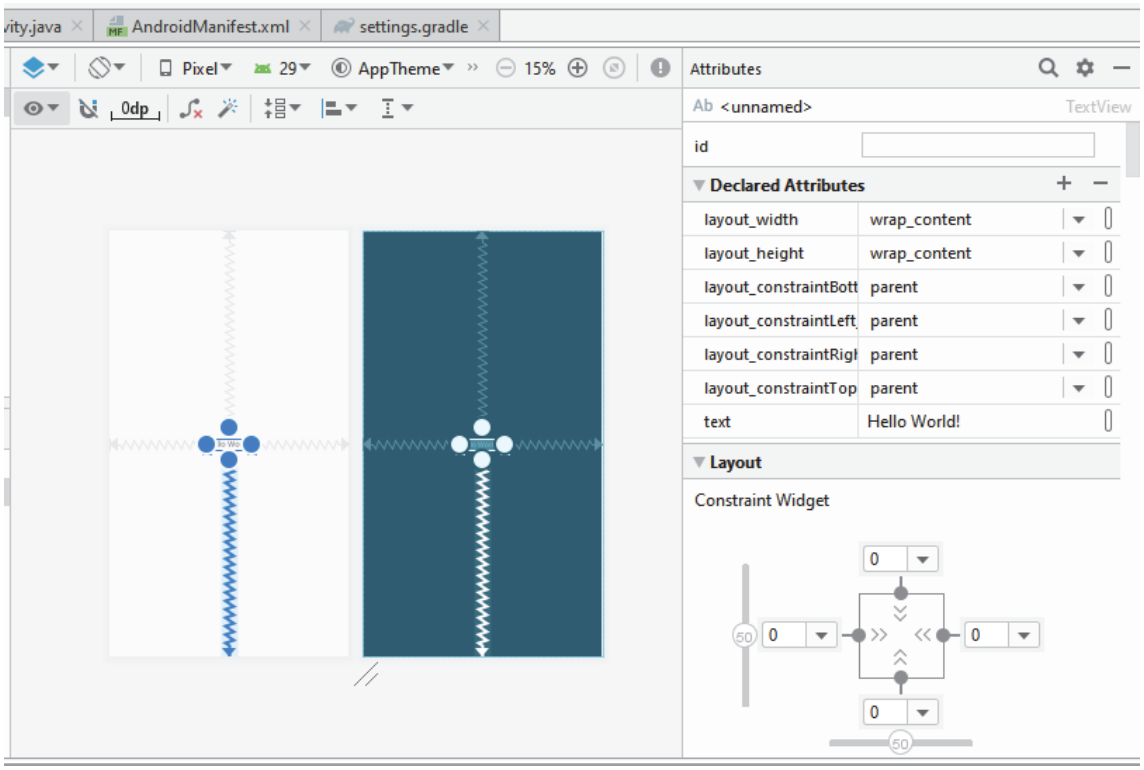


Рис. 1.6. Окно свойств

В методе *OnCreate*() идет обращение к методу родительского класса и установка ресурса разметки дизайна:

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

Чтобы установить ресурс разметки дизайна, вызывается метод *setContentView*, в который передается идентификатор ресурса. Идентификатор ресурса выглядит следуюшим образом: *R.layout.activity_main*. Это и есть ссылка на файл *activity_main.xml*, который находится в каталоге *res/layout*. Таким образом, при запуске приложения сначала запускается класс *MainActivity*, который в качестве графического интерфейса устанавливает разметку из файла *activity_main.xml*. Однако в классе *MainActivity* используются не файлы, а идентификаторы ресурсов: *R.layout.activity_main*.

Все идентификаторы ресурсов определены в классе *R*, который автоматически создается утилитой *appt* и находится в файле *R.java* в каталоге *build/r_class_sources/debug/r/...* (рис. 1.7). Класс *R* содержит идентификаторы для всех ресурсов. Для каждого типа ресурсов в классе *R* создается внутренний класс (например, для всех графических ресурсов из каталога *res/drawable* создается класс *R.drawable*) и каждому ресурсу присваивается идентификатор (рис. 1.7).

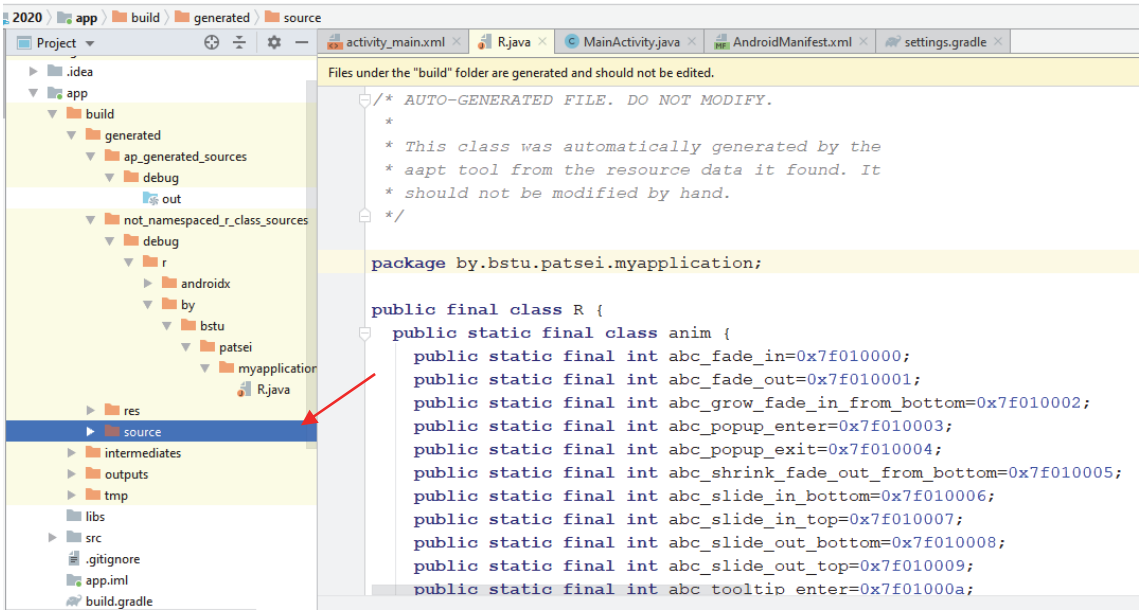


Рис. 1.7. Содержимое и физическое размещение файла ресурсов

По этому идентификатору впоследствии можно извлечь ресурс в файле кода. При обновлении ресурсов во время компиляции этот файл также обновляется.

1.1.4. Запуск приложения

Запускать приложение можно на реальном Android-устройстве или эмуляторе. При этом на эмуляторе можно выбрать одно из установленных виртуальных устройств, создать новый эмулятор (рис. 1.8) в Android Studio или использовать сторонние эмуляторы: Genymotion, Droid4x, BlueStacks и др.

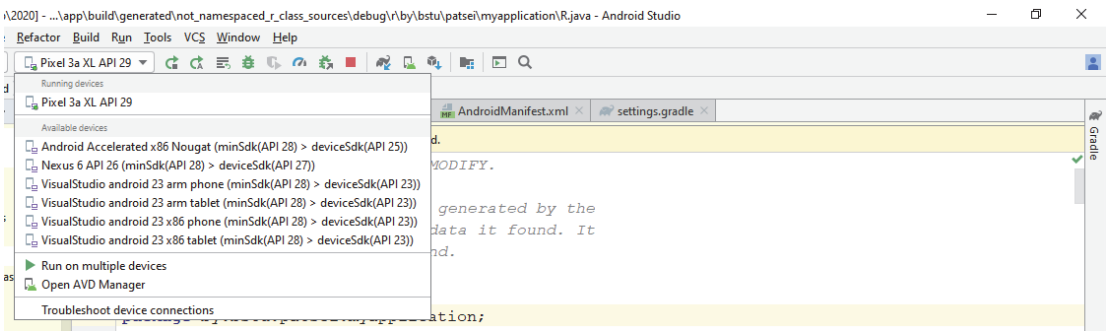


Рис. 1.8. Выбор места установки приложения

При использовании мобильного устройства для размещения приложения необходимо установить опции разработчика на мобильном устройстве. Для этого следует найти *Settings* → *About phone* (*Настройки* →

→ *О телефоне*) и семь раз нажать *Build Number* (*Номер сборки*). Вернуться к предыдущему экрану, на котором будет доступный пункт *Developer options* (*Для разработчика*). Перейти к пункту *Для разработчиков* и включить возможность отладки по USB. Через SDK Manager установить пакет Google Usb Driver или другой драйвер. После чего окно установки приложения отобразит подключенное устройство (рис. 1.9).

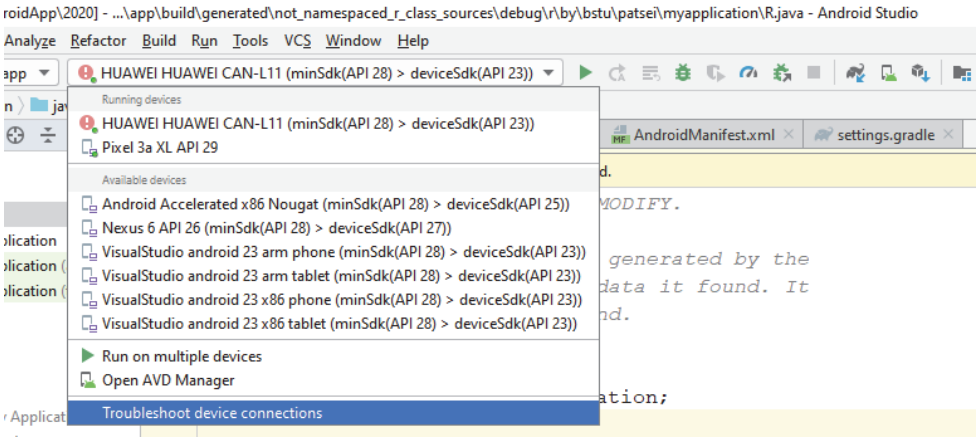


Рис. 1.9. Выбор места установки приложения

Результат запуска расмотренного приложения представлен на рис. 1.10.

Рис. 1.10. Результат запуска приложения

1.2. Архитектура Android и процесс компиляции мобильного приложения

1.2.1. Архитектура операционной системы Android

Основой платформы Android является ***ядро Linux***. Android Runtime (ART) полагается на ядро Linux для базовых функций, таких как потоковая обработка и управление памятью на низком уровне. Ядро Linux позволяет Android использовать ключевые функции безопасности и производителям устройств разрабатывать аппаратные драйверы (рис. 1.11).

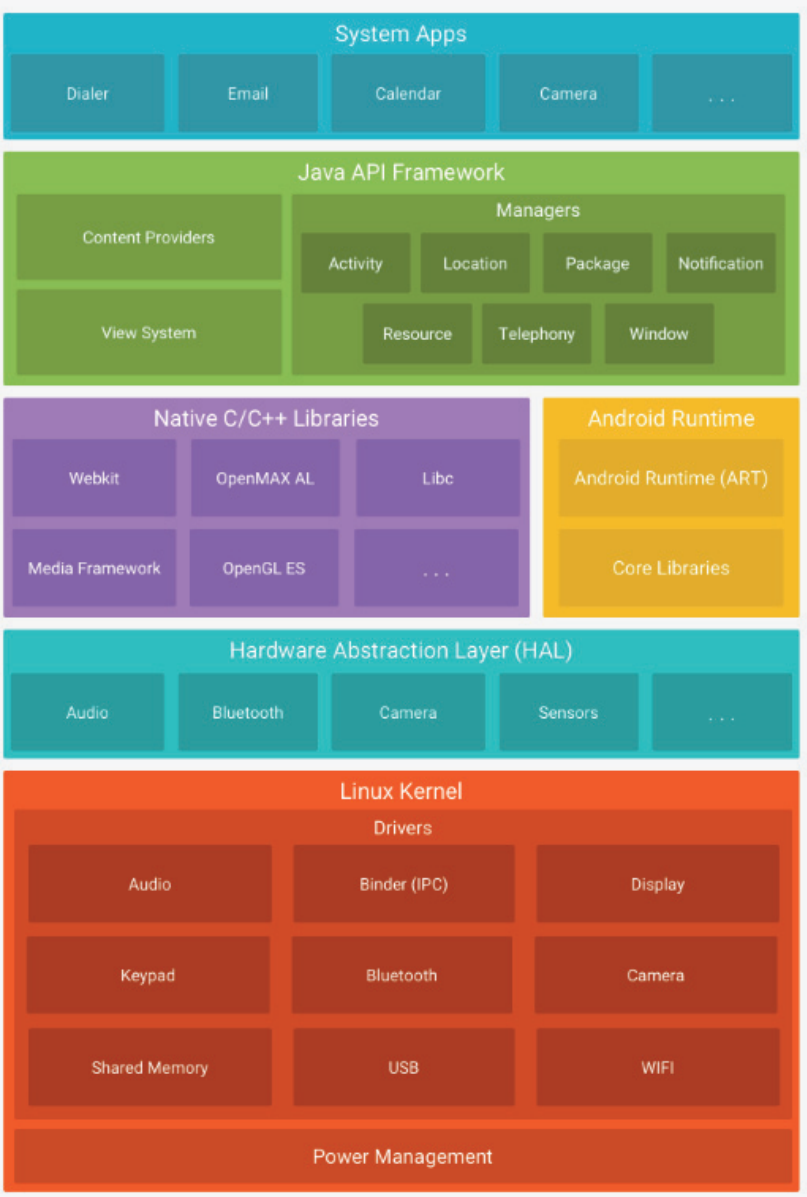


Рис. 1.11. Архитектура Android

Уровень абстракции аппаратного обеспечения (HAL) предоставляет стандартные интерфейсы. HAL состоит из нескольких библиотечных модулей, каждый из которых реализует интерфейс для определенного типа аппаратного компонента, такого как камера или модуль *bluetooth*.

Для устройств под управлением Android версии 5.0 (API уровня 21) или выше каждое приложение запускается в собственном процессе и с собственным экземпляром Android Runtime (ART). ART записывается для запуска нескольких виртуальных машин на устройствах путем выполнения DEX-файлов (формат байт-кода, разработанный специально для Android, который оптимизирован для минимального объема памяти).

До версии Android 5.0 (уровень API 21) Dalvik (регистровая виртуальная машина для выполнения программ, написанных на языке программирования Java, входит в операционную систему Android) была машиной времени исполнения Android. Android также включает в себя набор основных библиотек времени выполнения, которые обеспечивают большую часть функциональных возможностей языка программирования Java.

Многие базовые компоненты и сервисы Android, такие как ART и HAL, построены из собственного кода, для которого требуются библиотеки, написанные на C и C++. Платформа Android предоставляет API-интерфейсы Java Framework, чтобы выявить функциональность некоторых из этих родных библиотек для приложений. Например, доступ к OpenGL ES можно получить через Java OpenGL API платформы Android, чтобы добавить поддержку для рисования и управления 2D- и 3D-графикой в приложении.

Весь набор функций Android доступен через API, написанный на языке Java. Эти ***API-интерфейсы*** образуют строительные блоки, необходимые для создания Android-приложений, упрощая повторное использование компонентов и сервисов. Разработчики имеют полный доступ к тем же API-интерфейсам платформы, которые используют системные приложения Android.

Системные приложения Android поставляются с набором основных приложений для электронной почты, SMS-сообщений, календарей, просмотра контактов и т. д. Системные приложения функционируют как самостоятельные, а также предоставляют ключевые возможности, которые разработчики могут получать из своего собственного приложения.

Для применения новых возможностей языка Java необходимо также использовать набор инструментов *Jack*. С его помощью Android компилирует языковой источник Java в считываемый Android байт-код Dalvik Executable (DEX). В *Jack* предусмотрен собственный формат библиотеки *jack*.

1.2.2. Процесс сборки Android-приложения

Когда запускается сборка, первым делом читается файл *Android-Manifest.xml*, в нем есть важные параметры, такие как *package* (например, by.belstu.app) и *targetSdkVersion*.

Затем вызывается программа *aapt* (Android Asset Packaging Tool), которой передается *AndroidManifest.xml*, папка с ресурсами *res/, assets/,* путь к *android.jar* нужной target-версии. Программа *aapt* проверяет ресурсы и компилирует их, создавая при этом класс *R.java*, содержащий идентификаторы ресурсов и файл *resources.arsc*, в котором имеется информация об XML-ресурсах и их атрибутах (рис. 1.12).

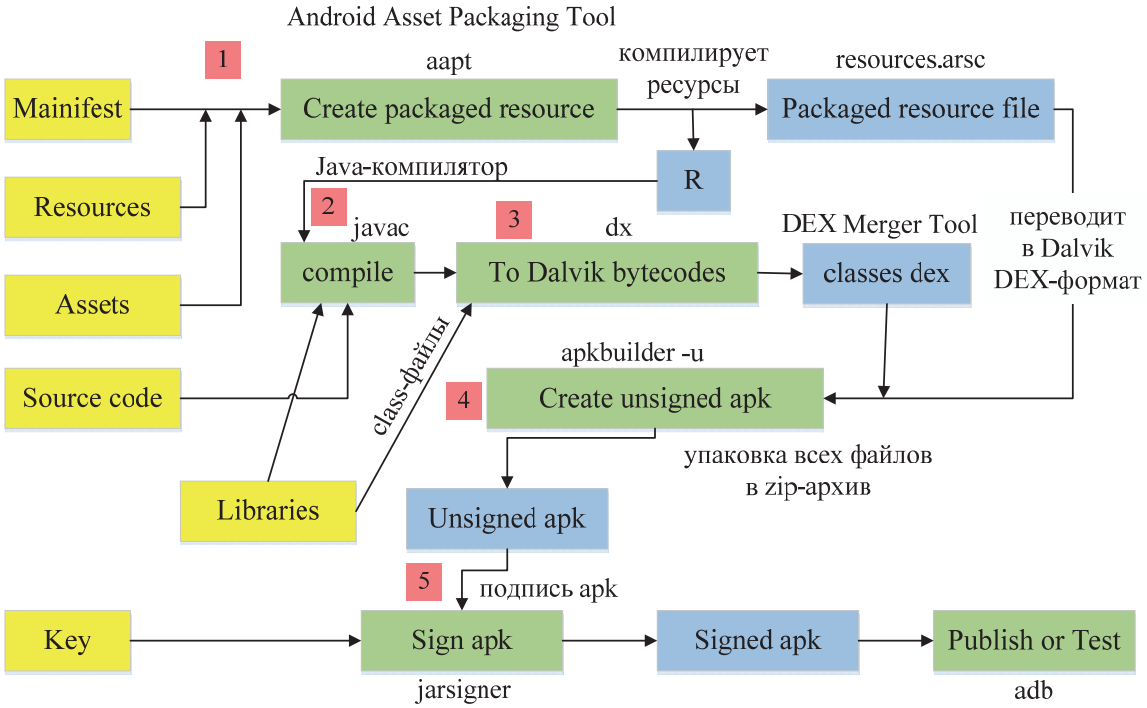


Рис. 1.12. Процесс сборки Android-приложения

Далее подхватываются все библиотеки, которые используются в проекте и запускается Java-компилятор *javac*. Полученные *class*-файлы передаются в программу *dx*, которая переводит их в DEX-формат. Причем для оптимизации готовые библиотеки дексируются отдельно,

а классы проекта отдельно (оптимизация в том, что индексированные библиотеки можно закешировать). Если собралось несколько DEX-файлов, то все они объединяются при помощи *DEX Merger Tool*. В конечном итоге получается единственный файл *classes.dex* (или несколько, если используется *multidex*) (рис. 1.12). Теперь есть все компоненты и можно собирать apk. По сути, это просто упаковка всех файлов в zip-архив, но используется специальная программа *apkbuilder*. После ее выполнения получаем неподписанный apk-файл, то есть без папки META-INF внутри.

Последний этап – подпись apk. Берется заранее сгенерированный ключ и передается в *jarsigner* вместе с неподписанным apk-файлом. На выходе имеем готовое приложение, которое можно устанавливать (рис. 1.12).

2. СОЗДАНИЕ ИНТЕРФЕЙСА. МАКЕТЫ. ЭЛЕМЕНТЫ UI. РЕСУРСЫ

2.1. Организация пользовательского интерфейса

Все элементы интерфейса в приложении Android создаются с помощью объектов *View* и *ViewGroup*. Объект *View* формирует на экране элемент, с которым пользователь может взаимодействовать. Объект *ViewGroup* содержит другие объекты *View* (и *ViewGroup*) для определения макета интерфейса.

Android предоставляет коллекцию подклассов *View* и *ViewGroup*, которая включает в себя обычные элементы ввода и различные модели макета. Каждая группа представляет собой невидимый контейнер, в котором объединены дочерние виды. Эта древовидная иерархия может быть простой или сложной (чем проще, тем лучше для производительности).

Для отладки макетов можно воспользоваться инструментом *Hierarchy Viewer*. С его помощью можно просмотреть значения свойств, рамки с индикаторами заполнения или поля, а также полностью отрисованные представления прямо во время отладки приложения на эмуляторе или на устройстве.

2.1.1. Разработка Layout

Макет определяет визуальную структуру пользовательского интерфейса. Существует два способа объявить макет (рис. 2.1):

— объявление элементов пользовательского интерфейса в XML;

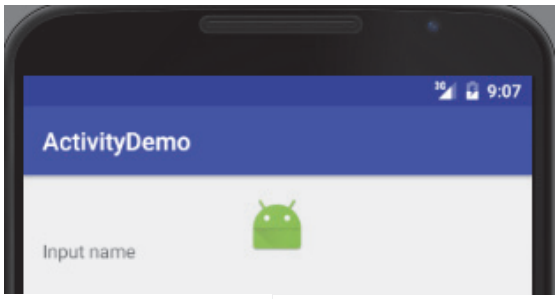
— создание экземпляров элементов во время выполнения (приложение может программным образом создавать объекты *View* и *ViewGroup*).

В приложениях Android визуальный интерфейс загружается из специальных файлов XML, которые хранят разметку. Эти файлы являются ресурсами разметки.

Объявление пользовательского интерфейса в файлах XML позволяет отделить интерфейс приложения от кода. В приложении могут быть определены разметки в файлах XML для различных ориентаций монитора, размеров устройств, языков и т. д. Кроме того, объявление разметки в XML позволяет легче визуализировать структуру интерфейса и облегчает отладку (рис. 2.2).

Активность – специальный класс Java, который решает, какой макет следует использовать, и описывает, как приложение должно реагировать на действия пользователя

Макеты могут включать компоненты графических интерфейсов: кнопки, текстовые поля, подписи и т. д.




```
public class MainActivity extends AppCompatActivity {

    public void functionPress(View view) {

        EditText username = (EditText)findView-
ById(R.id.userName);
        Log.i("Username",
username.getText().toString());
        Toast.makeText(getApplicationCon-
text(),username.getText().toString(),
Toast.LENGTH_LONG).show();

        ImageView imageView = (Image-View)findView-
ById(R.id.imageIcon);
        imageView.setImageResource
R.drawable.penguins);
    }

    @Override
    protected void onCreate(Bundle savedInstance-
State) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.acti-vity_main);
    }
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.an-
droid.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="by.bstu.pnv.activitydemo.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Input name"
        android:id="@+id/textView"
        android:textSize="15sp"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_marginTop="35dp" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/userName"
        android:textSize="15sp"
        android:layout_below="@+id/textView"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_marginTop="47dp" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Enter"
        android:id="@+id/button"
        android:layout_below="@+id/userName"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_marginTop="57dp"
        android:onClick="functionPress" />

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/imageIcon"
        android:src="@android:drawable/sym_def_app_icon"
        android:scaleType="fitXY"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true" />
</RelativeLayout>
```

Рис. 2.1. Разработка макета пользовательского интерфейса Android-приложения

Файлы разметки графического интерфейса располагаются в проекте в каталоге *res/layout*.

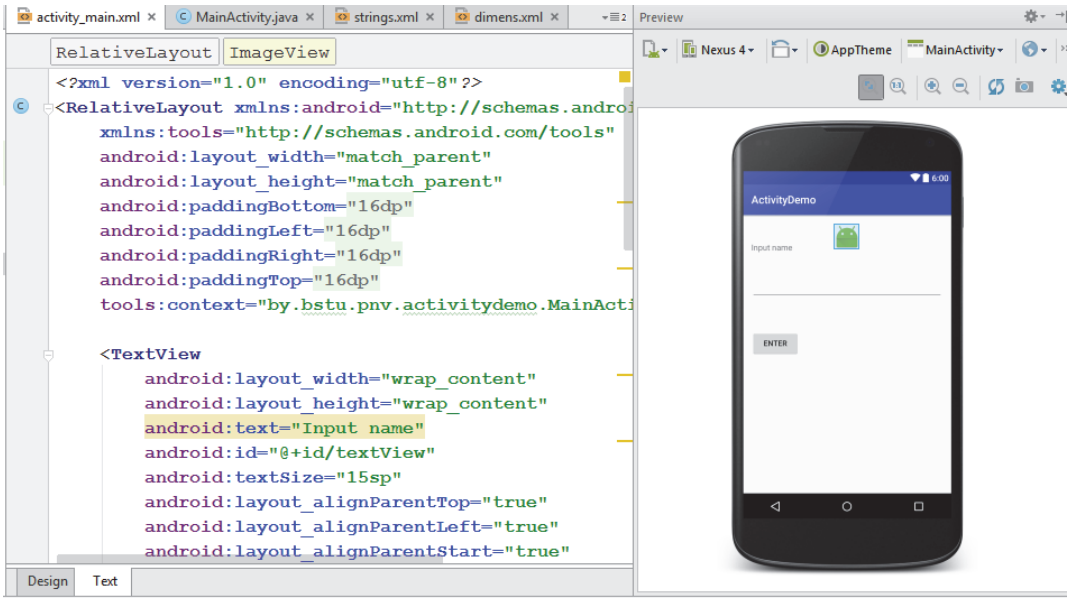


Рис. 2.2. Разработка макета пользовательского интерфейса на основе XML

2.1.2. Разработка интерфейса в режиме дизайнера

Android Studio имеет инструментарий, который облегчает разработку графического интерфейса. Можно открыть файл XML и с помощью кнопки *Design* переключиться в режим дизайнера к графическому представлению интерфейса в виде эскиза устройства (рис. 2.3).

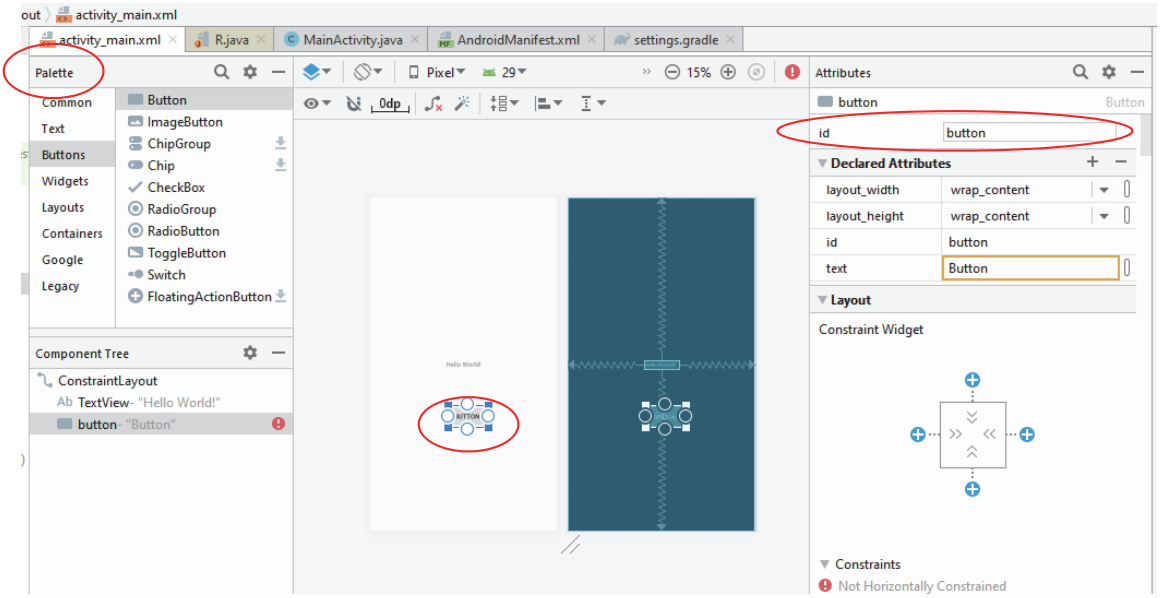


Рис. 2.3. Работа в режиме *Design*

Слева будет находиться панель инструментов, из которой можно перенести нужный элемент мышкой на эскиз. Все перенесенные элементы будут автоматически добавляться в файл XML.

При выделении элемента справа появится окно *Properties* – панель свойств выделенного элемента. Здесь можно изменить значения свойств элемента.

2.2. Установка размеров

В операционной системе Android можно использовать различные типы измерений:

— *px* – пиксели текущего экрана. Эта единица измерения не рекомендуется, так как реальное представление внешнего вида может изменяться в зависимости от устройства;

— *dp* (device-independent pixels) – независимые от плотности экрана пиксели. Абстрактная единица измерения, основанная на физической плотности экрана с разрешением 160 dpi (точек на дюйм). В этом случае 1 dp = 1 px. Если размер экрана больше или меньше, чем 160 dpi, количество пикселей, которые применяются для отрисовки 1 dp, соответственно увеличивается или уменьшается. Общая формула для получения количества физических пикселей из dp: px = dp · (dpi / 160);

— *sp* (scale-independent pixels) – независимые от масштабирования пиксели. Допускают настройку размеров, производимую пользователем. Рекомендуются для работы со шрифтами;

— *pt* – 1/72 дюйма, величина базируется на физических размерах экрана;

— *mm* – миллиметры;

— *in* – дюймы.

Предпочтительными единицами для использования являются dp.

Для упрощения работы с размерами все они разбиты на несколько групп:

— ldpi (low) ~120 dpi;
— mdpi (medium) ~160 dpi;
— hdpi (high) ~240 dpi;
— xhdpi (extra-high) ~320 dpi;
— xxhdpi (extra-extra-high) ~480 dpi;
— xxxhdpi (extra-extra-extra-high) ~640 dpi.

Все визуальные элементы, которые используются в приложении, как правило, упорядочиваются на экране с помощью контейнеров.

В Android подобными контейнерами служат классы *RelativeLayout*, *LinearLayout*, *GridLayout*, *TableLayout*, *ConstraintLayout*, *FrameLayout* и др. Все они по-разному располагают элементы и управляют ими, но есть некоторые общие моменты при компоновке визуальных компонентов.

Для организации элементов внутри контейнера используются параметры разметки. Для их задания в файле XML используются атрибуты, которые начинаются с префикса *layout_*. К таким параметрам относятся атрибуты *layout_height* и *layout_width*, которые используются для установки размеров и могут принимать одно из следующих значений:

— *точные размеры* элемента, например 96 dp (рис. 2.4);

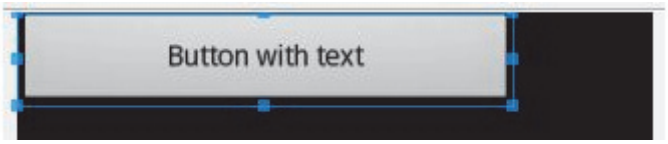


Рис. 2.4. Определение элемента
с точными размерами

— *значение wrap_content*: элемент растягивается до тех границ, которые достаточны, чтобы вместить все его содержимое (рис. 2.5);

Рис. 2.5. Определение элемента в соответствии
с содержимым

— *значение match_parent*: элемент заполняет всю область родительского контейнера (рис. 2.6).

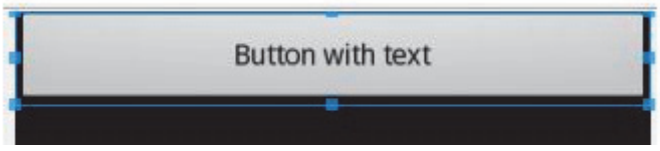


Рис. 2.6. Определение элемента в соответствии
с родительским контейнером

Можно дополнительно ограничить минимальные и максимальные значения с помощью атрибутов *minWidth*/*maxWidth* и *minHeight*/*maxHeight*:

```
android:minWidth="200dp"
android:maxWidth="250dp"
android:minHeight="100dp"
android:maxHeight="200dp"
android:layout_height="wrap_content"
android:layout_width="wrap_content"
```

Если элемент создается в коде Java, то для установки высоты и ширины можно использовать метод *setLayoutParams*():

```
  TextView textView1 = new TextView(this);
      textView1.setText("Hello Android");
      textView1.setTextSize(26);

  // Устанавливаем размеры
textView1.setLayoutParams(new ViewGroup.LayoutParams
        (ViewGroup.LayoutParams.WRAP_CONTENT,
         ViewGroup.LayoutParams.WRAP_CONTENT));
```

Параметры разметки позволяют задать отступы как от внешних границ элемента до границ контейнера, так и внутри самого элемента между его границами и содержимым.

Для установки внутренних отступов применяется атрибут *android*:*padding*. Он устанавливает отступы контента от всех четырех сторон контейнера. Можно устанавливать отступы только от одной стороны контейнера, применяя атрибуты: *android*:*paddingLeft*, *android*:*paddingRight*, *android*:*paddingTop* и *android*:*paddingBottom* (рис. 2.7).

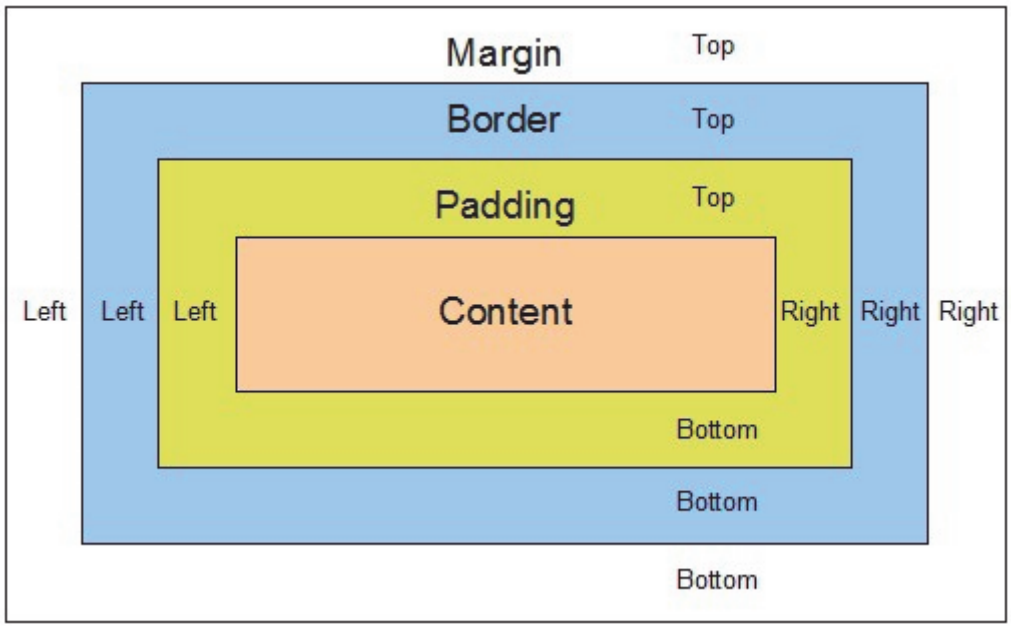


Рис. 2.7. Отступы элементов управления

Для установки внешних остступов используется атрибут *layout_margin*. Он имеет модификации, которые позволяют задать отступ только от одной стороны: *android*:*layout_marginBottom*, *android*:*layout_marginTop*, *android*:*layout_marginLeft* и *android*:*layout_marginRight* (рис. 2.7):

```
android:layout_marginTop="50dp"
android:layout_marginBottom="60dp"
android:layout_marginLeft="60dp"
android:layout_marginRight="60dp"
```

Для программной установки внутренних отступов вызывается метод *setPadding*(*left*, *top*, *right*, *bottom*), в который передаются четыре значения для каждой из сторон.

2.3. Виды Layout

2.3.1. LinearLayout

Контейнер *LinearLayout* представляет объект *ViewGroup*, который упорядочивает все дочерние элементы в одном направлении: по горизонтали или по вертикали. Все элементы расположены один за другим. Направление разметки указывается с помощью атрибута *android*:*orientation*. Пример отображения приведенной ниже разметки представлен на рис. 2.8.

```
<?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">
    <Button
        android:id="@+id/Button01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/app_name"/>
    <Button
        android:id="@+id/Button02"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/enter"/>
    <Button
        android:id="@+id/Button04"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/greeting"/>
    <TextView
        android:id="@+id/TextView01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/greeting"/>
</LinearLayout>
```

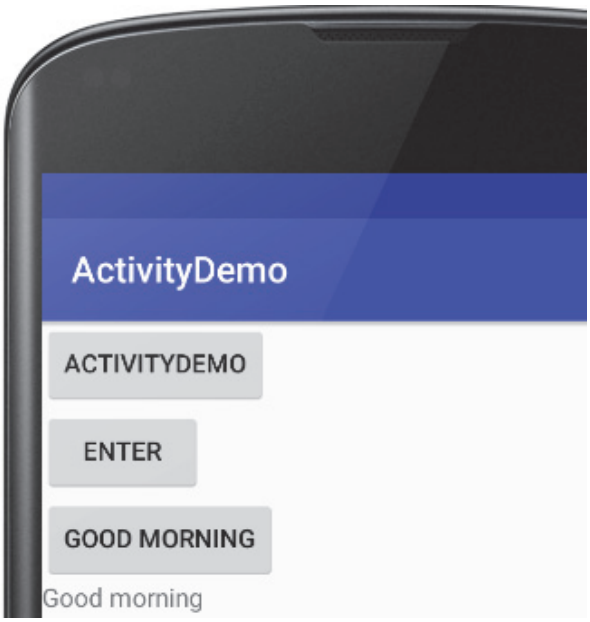


Рис. 2.8. Пример интерфейса
с контейнером *LinearLayout*

LinearLayout поддерживает свойство – вес элемента, которое передается атрибутом *android*:*layout_weight*. Это свойство принимает значение, указывающее, какую часть оставшегося свободного места контейнера по отношению к другим объектам займет данный элемент. Например, если один элемент будет иметь для свойства *android*:*layout_weight* значение 2, а другой – значение 1, то в сумме они дадут 3, поэтому первый элемент будет занимать 2/3 оставшегося пространства, а второй – 1/3. Если все элементы имеют значение *android*:*layout_weight*="1", то они будут равномерно распределены по всей площади контейнера.

2.3.2. RelativeLayout

RelativeLayout представляет объект *ViewGroup*, который располагает дочерние элементы относительно позиции других дочерних элементов разметки или относительно области самой разметки *RelativeLayout*. Используя относительное позиционирование, можно установить элемент по правому краю, в центре или другим способом. В настоящее время он считается устаревшим и располагается в папке Legacy.

Для установки элемента в файле XML применяются следующие атрибуты:

– *android:layout_above* – располагает элемент над элементом с указанным ID;

– *android:layout_below* – располагает элемент под элементом с указанным ID;

– *android:layout_toLeftOf* – располагает элемент слева от элемента с указанным ID;

– *android:layout_toRightOf* – располагает элемент справа от элемента с указанным ID;

– *android:layout_alignBottom* – выравнивает элемент по нижней границе другого элемента с указанным ID;

– *android:layout_alignLeft* – выравнивает элемент по левой границе другого элемента с указанным ID;

– *android:layout_alignRight* – выравнивает элемент по правой границе другого элемента с указанным ID;

– *android:layout_alignTop* – выравнивает элемент по верхней границе другого элемента с указанным ID;

– *android:layout_alignBaseline* – выравнивает базовую линию элемента по базовой линии другого элемента с указанным ID;

– *android:layout_alignParentBottom* – если атрибут имеет значение *true*, то элемент прижимается к нижней границе контейнера;

– *android:layout_alignParentRight* – если атрибут имеет значение *true*, то элемент прижимается к правому краю контейнера;

– *android:layout_alignParentLeft* – если атрибут имеет значение *true*, то элемент прижимается к левому краю контейнера;

– *android:layout_alignParentTop* – если атрибут имеет значение *true*, то элемент прижимается к верхней границе контейнера;

– *android:layout_centerInParent* – если атрибут имеет значение *true*, то элемент располагается по центру родительского контейнера;

– *android:layout_centerHorizontal* – при значении *true* элемент выравнивается по центру по горизонтали;

– *android:layout_centerVertical* – при значении *true* элемент выравнивается по центру по вертикали.

Пример отображения разметки, представленной ниже, приведен на рис. 2.9.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/an-
droid"
    android:layout_width="match_parent" android:lay-
out_height="match_parent">

        <Button
         android:id="@+id/button_center"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:height="20pt"
         android:longClickable="true"
         android:text="@string/greeting"
```

```
        android:layout_centerVertical="true"
        android:layout_centerInParent="true"/>

    <Button
        android:id="@+id/button_next"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerVertical="true"
        android:layout_marginLeft="30sp"
        android:height="20pt"
        android:longClickable="true"
        android:text="@string/enter"
        android:width="20pt"/>

    <Button
        android:id="@+id/button_right"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBaseline="@+id/button_center"
        android:layout_alignBottom="@+id/button_center"
        android:layout_alignParentRight="true"
        android:layout_alignWithParentIfMissing="false"
        android:text="Next"
        android:layout_marginRight="30sp" />
    </RelativeLayout>
```

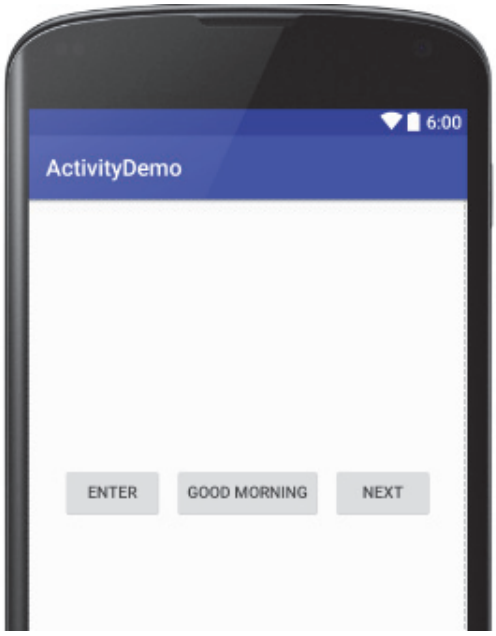


Рис. 2.9. Пример интерфейса
с контейнером *RelativeLayout*

Для управления позиционированием элемента при определении интерфейса можно использовать атрибуты *gravity* и *layout_gravity*. Атрибут *gravity* задает позиционирование содержимого внутри объекта. Он может принимать следующие значения:

— *top* – элемент размещается вверху;
— *bottom* – элемент размещается внизу;
— *left* – элемент размещается в левой стороне;
— *right* – элемент размещается в правой стороне контейнера;
— *center_vertical* – выравнивает элементы по центру по вертикали;
— *center_horizontal* – выравнивает элементы по центру по горизонтали;
— *center* – элемент размещается по центру;
— *fill_vertical* – элемент растягивается по вертикали;
— *fill_horizontal* – элемент растягивается по горизонтали;
— *fill* – элемент заполняет все пространство контейнера;
— *clip_vertical* – обрезает верхнюю и нижнюю границы элементов;
— *clip_horizontal* – обрезает правую и левую границы элементов;
— *start* – элемент позиционируется в начале контейнера (в верхнем левом углу);
— *end* – элемент позиционируется в конце контейнера (в верхнем правом углу).

2.3.3. TableLayout

Контейнер *TableLayout* структурирует элементы по столбцам и строкам. Android находит строку с максимальным количеством виджетов одного уровня, и это количество будет означать количество столбцов. Если бы в какой-нибудь из них было три виджета, то, соответственно, столбцов было бы также три, даже если в другой строке осталось бы два виджета. Например, определены три строки, в каждой из которых изменяется два или три элемента (рис. 2.10):

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:lay-
out_height="match_parent">
   <TableRow>
       <Button />
       <Button />
       <Button />
   </TableRow>
   <TableRow>
      <Button />
      <Button />
   </TableRow>
   <TableRow>
      <Button />
      <Button />
      <Button />
   </TableRow>
</TableLayout>
```

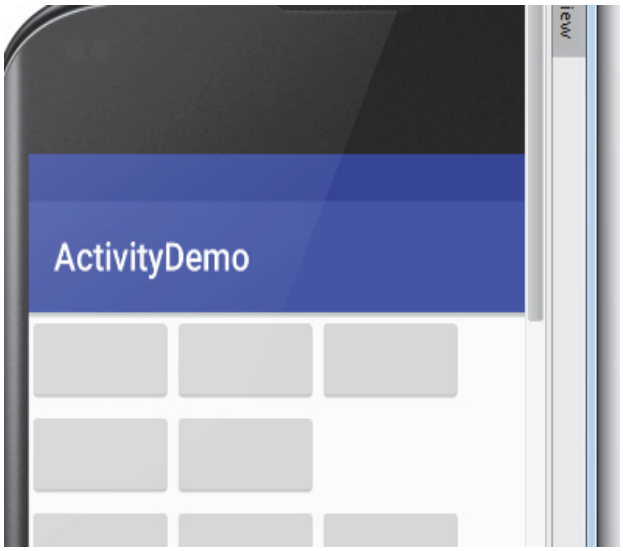


Рис. 2.10. Пример интерфейса
с контейнером *TableLayout*

Элемент *TableRow* наследуется от класса *LinearLayout*, поэтому можно к нему применять тот же функционал, что и к *LinearLayout*.

Если какой-то элемент должен быть растянут на ряд столбцов, то это выполняется с помощью атрибута *layout_column*, который указывает, на какое количество столбцов надо растянуть элемент.

Также элемент можно растянуть на всю строку, установив атрибут *android*:*layout_weight*="1".

2.3.4. FrameLayout

Контейнер *FrameLayout* предназначен для вывода на экран одного помещенного в него визуального элемента. Если поместить несколько элементов, то они будут накладываться друг на друга (рис. 2.11):

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/an-
droid" android:layout_width="match_parent" android:lay-
out_height="match_parent">

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />

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

Рис. 2.11. Пример интерфейса
с контейнером *FrameLayout*

Элементы управления, которые помещаются в контейнер *FrameLayout*, могут установить свое позиционирование с помощью атрибута *android*:*layout_gravity* (рис. 2.12).

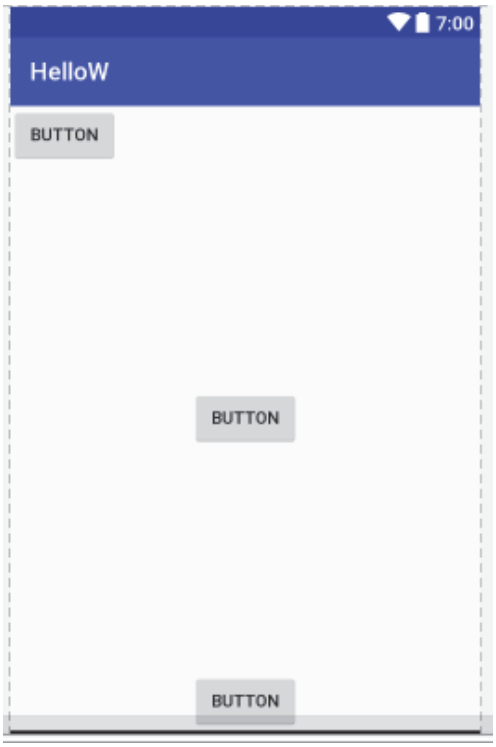


Рис. 2.12. Пример интерфейса
с контейнером *FrameLayout*
и атрибутом *android*:*layout_gravity*

При указании значения атрибута можно комбинировать ряд значений, разделяя их вертикальной чертой *bottom*|*center_horizontal:*

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent">
    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:layout_gravity="top"
        android:text="Button" />
    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="Button" />
    <Button
        android:id="@+id/button3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|center_horizontal"
        android:text="Button" />
</FrameLayout>
```

2.3.5. GridLayout

GridLayout – это контейнер, который позволяет создавать табличные представления. В настоящее время он считается устаревшим и располагается в папке Legacy.

GridLayout состоит из коллекции строк, каждая из которых состоит из отдельных ячеек. С помощью атрибутов *android*:*rowCount* и *android*:*columnCount* устанавливается число строк и столбцов соответственно. *GridLayout* автоматически может позиционировать вложенные элементы управления по строкам. При этом ширина столбцов устанавливается автоматически по ширине самого широкого элемента. В слудеющем примере устанавливается 2 строки и 3 столбца, первая кнопка попадает в первую ячейку (первая строка первый столбец), вторая кнопка – во вторую ячейку и так далее (рис. 2.13):

```
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:rowCount="2"
    android:columnCount="3">

    <Button android:text="1" />
    <Button android:text="2" />
    <Button android:text="3" />
    <Button android:text="4" />
    <Button android:text="5" />
    <Button android:text="6" />
    <Button android:text="7" />
    <Button android:text="8" />
    <Button android:text="9" />
    <Button android:text ="sin"/>
    <Button android:text ="cos"/>
    <Button android:text ="/"/>
    <Button android:text ="*"/>
</GridLayout>
```

Рис. 2.13. Пример интерфейса
с контейнером *FrameLayout*

Можно явно задать номер столбца и строки для определенного элемента, а при необходимости растянуть на несколько столбцов или строк. Для этого используют атрибуты:

– *android*:*layout_column* – номер столбца (отсчет идет от нуля);

– *android*:*layout_row* – номер строки;

– *android*:*layout_columnSpan* – количество столбцов, на которые растягивается элемент;

– *android*:*layout_rowSpan* – количество строк, на которые растягивается элемент.

2.3.6. ConstraintLayout

ConstraintLayout – относительно новый тип контейнера, который является развитием *RelativeLayout* и позволяет создавать гибкие и масштабируемые интерфейсы.

Для позиционирования элемента внутри *ConstraintLayout* необходимо указать ограничения (constraints). Есть несколько типов ограничений. Для установки позиции относительно определенного элемента используются следующие ограничения:

– *layout_constraintLeft_toLeftOf* – левая граница позиционируется относительно левой границы другого элемента (аналог *layout_constraintStart_toStartOf*);

– *layout_constraintLeft_toRightOf* – левая граница позиционируется относительно правой границы другого элемента (аналог *layout_constraintStart_toEndOf*);

— *layout_constraintRight_toLeftOf* – правая граница позиционируется относительно левой границы другого элемента (аналог *layout_constraintEnd_toStartOf*);

— *layout_constraintRight_toRightOf* – правая граница позиционируется относительно правой границы другого элемента (аналог *layout_constraintEnd_toEndOf*);

— *layout_constraintTop_toTopOf* – верхняя граница позиционируется относительно верхней границы другого элемента;

— *layout_constraintBottom_toBottomOf* – нижняя граница позиционируется относительно нижней границы другого элемента;

— *layout_constraintBaseline_toBaselineOf* – базовая линия позиционируется относительно базовой линии другого элемента;

— *layout_constraintTop_toBottomOf* – верхняя граница позиционируется относительно нижней границы другого элемента;

— *layout_constraintBottom_toTopOf* – нижняя граница позиционируется относительно верхней границы другого элемента.

Позиционирование может производиться относительно границ самого контейнера *ContentLayout* (в этом случае ограничение имеет значение *parent*) либо же относительно любого другого элемента внутри *ConstraintLayout*, тогда в качестве значения ограничения указывается *id* этого элемента.

Чтобы указать отступы от элемента, относительно которого производится позиционирование, применяются следующие атрибуты:

— *android*:*layout_marginLeft* – отступ от левой границы;

— *android*:*layout_marginRight* – отступ от правой границы;

— *android*:layout_*marginTop* – отступ от верхней границы;

— *android*:*layout_marginBottom* – отступ от нижней границы;

— *android*:*layout_marginStart* – отступ от левой границы;

— *android*:*layout_marginEnd* – отступ от правой границы.

Если выделить на экране *Button*, то можно видеть 4 круга по его бокам (рис. 2.14). Эти круги используются, чтобы создавать привязки.

Существует два типа привязок. Одни задают положение *View* по горизонтали, а другие – по вертикали.

Чтобы создать горизонтальную привязку, нужно привязать положение *Button* к левому краю его родителя. Родителем *Button* является *ConstraintLayout*, который в нашем случае занимает весь экран. Поэтому края *ConstraintLayout* совпадают с краями экрана. Чтобы создать привязку, следует нажать мышкой на *Button* и выделить ее. Затем зажать левой кнопкой мыши левый кружок и тащить его к левой границе (рис. 2.14). *Button* также сдвигается влево. Она привязывается к левой границе своего родителя.

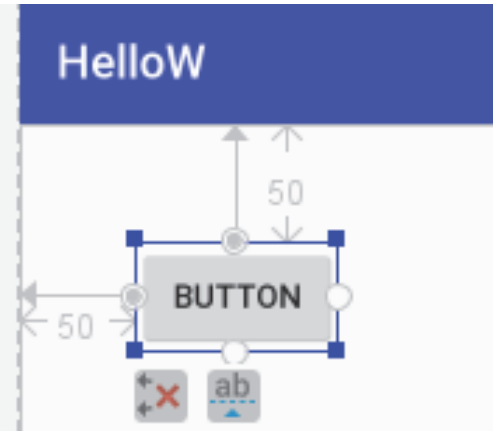


Рис. 2.14. Установка привязок
к границам контейнера

Чтобы закрепить *Button* и по вертикали, можно создать вертикальную привязку. Теперь *View* привязан и по горизонтали, и по вертикали (рис. 2.14). То есть он точно знает, где он должен находиться на экране во время работы приложения.

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="50dp"
    android:layout_marginTop="50dp"
    android:text="Button"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Можно привязываться не только к границам родителя, но и к другим *View*. Например, привяжем кнопку к *TextView*. Так как кнопка привязана к *TextView*, то при его перемещении, кнопка также будет перемещаться (рис. 2.15).

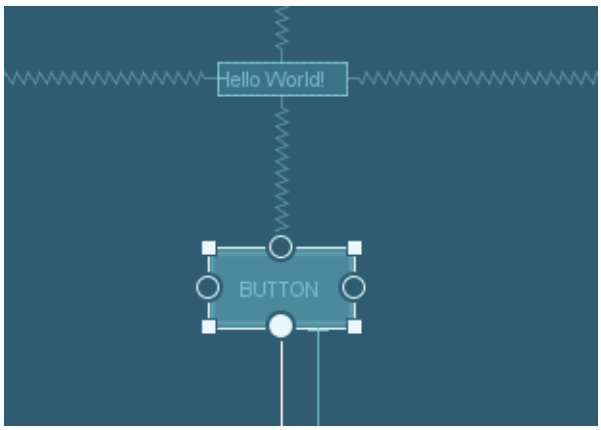


Рис. 2.15. Установка привязок к *View*

Чтобы удалить привязку, надо просто нажать на нее и выбрать *Clear* (рис. 2.16).

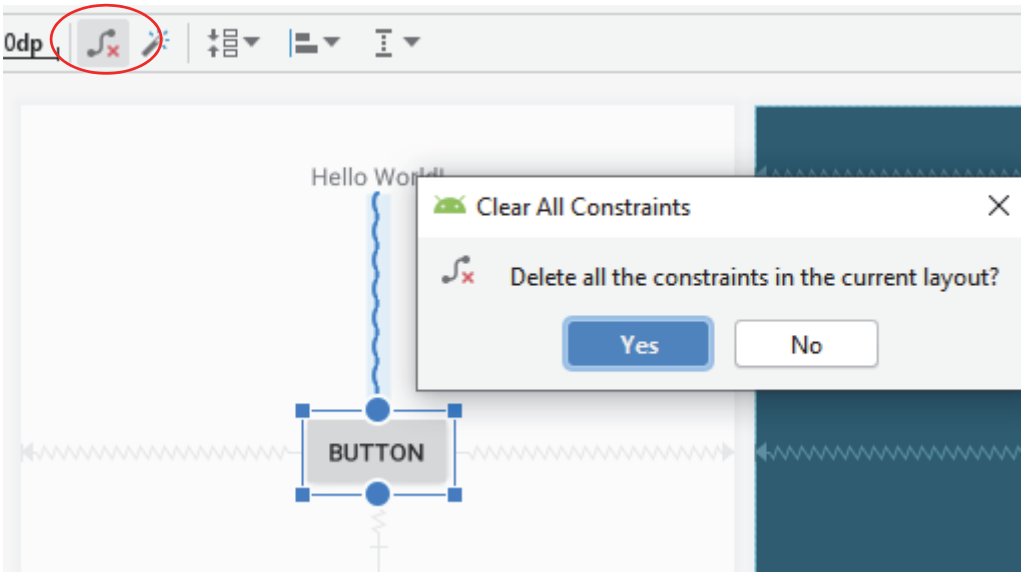


Рис. 2.16. Удаление привязки

Привяжем кнопку к левой и правой границам родителя. *Button* сначала уйдет влево, так как была сделана привязка к левой границе, но после создания привязки к правой границе она выровняется и теперь будет располагаться по центру (рис. 2.17). То есть привязки уравнивают друг друга, и *View* будет находиться ровно посередине между тем, к чему он привязан слева, и тем, к чему он привязан справа. Следует обратить внимание, что такие двусторонние привязки отображаются как пружинки, а не линии.

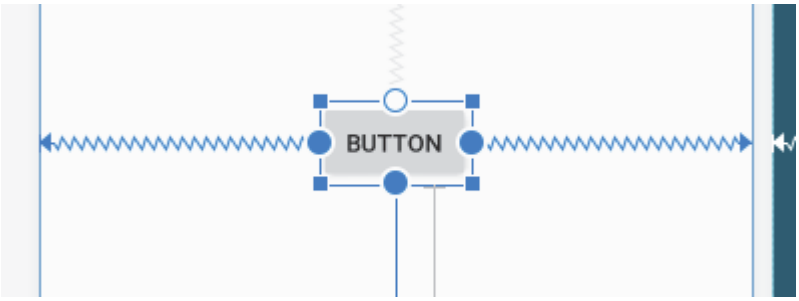


Рис. 2.17. Привязка к левой и правой границам

При создании ограничений необходимо придерживаться следующих правил:

— у каждого *View* должно быть как минимум два ограничения: одно горизонтальное и одно вертикальное;

— можно создавать ограничения только между дескриптором ограничения и точкой привязки, которые используют одну и ту же плоскость. Таким образом, вертикальная плоскость (левая и правая стороны) *View* может быть ограничена только другой вертикальной плоскостью и базовые линии могут ограничивать только другие базовые линии;

— каждое правило может быть использовано только для одного ограничения, но можно создавать несколько ограничений (с различных точек зрения) к одной и той же точке привязки.

На панели есть несколько инструментов, которые могут помочь в работе (рис. 2.18) (перечислены слева направо).

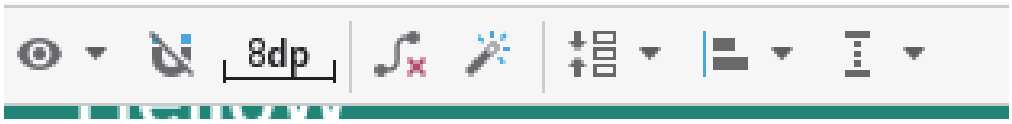


Рис. 2.18. Панель инструментов для работы с привязками

Показать/Скрыть привязки – если включено, то все привязки будут видны на экране, если выключено, то видны будут только привязки выделенного *View*.

Автопривязки – если включено, то можно создавать привязки к родителю.

Отступ – здесь можно автоматически задать, какой отступ будет использован по умолчанию при создании привязки.

Удалить все привязки – при нажатии этой кнопки все привязки на экране будут удалены.

Создать привязки – создает привязки для всех *View* на экране.

Собрать/Растянуть – собирает вместе несколько выделенных *View* сначала по горизонтали затем по вертикали. Эта операция не создает никаких привязок (рис. 2.19).

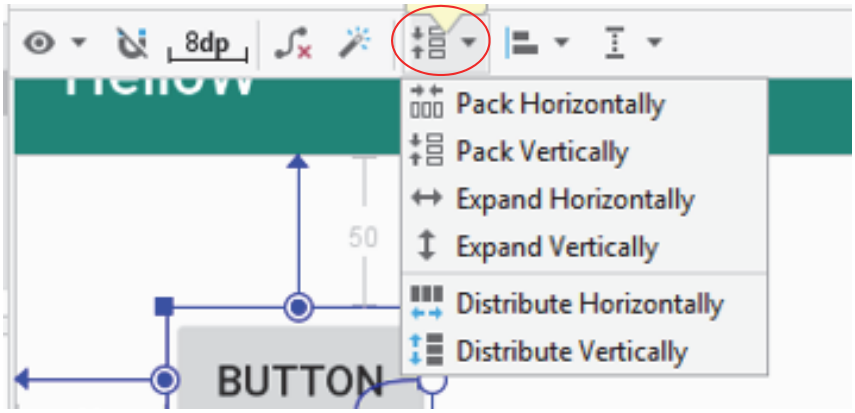


Рис. 2.19. Элемент *Собрать/Растянуть* на панели инструментов

Выравнивание – по горизонтали: по левому краю, по центру, по правому краю. Нижний ряд кнопок – это центрирование. Оно создает двустороннюю привязку (рис. 2.20).

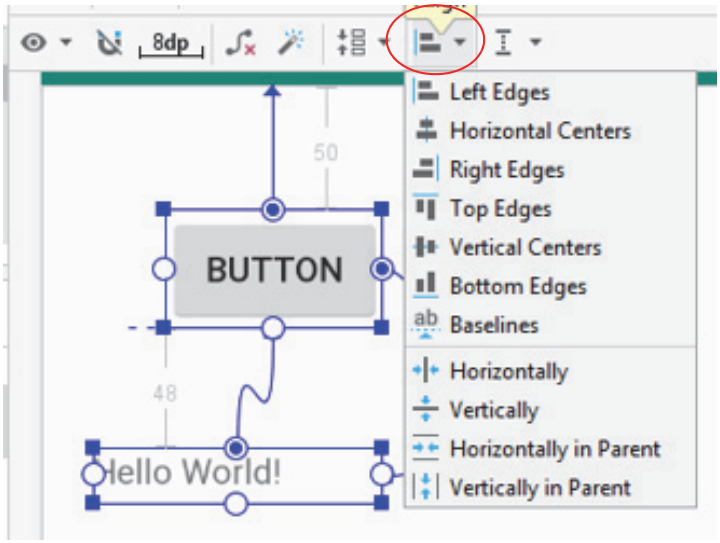


Рис. 2.20. Элемент *Выравнивание* на панели инструментов

Направляющие – это линии, которые можно использовать для создания привязок (рис. 2.21).

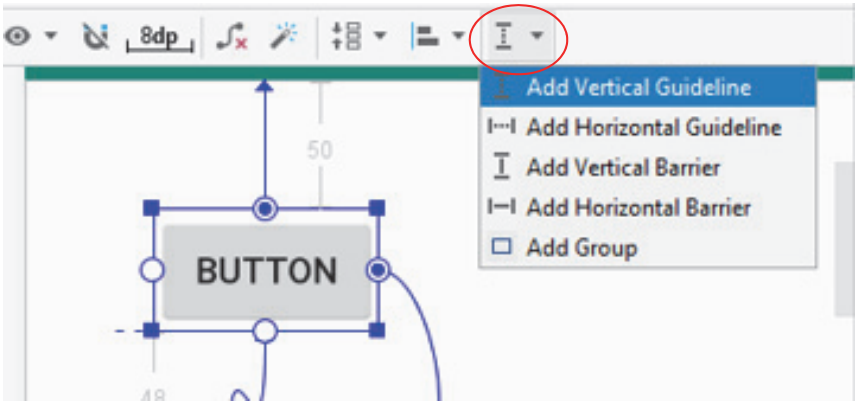


Рис. 2.21. Элемент *Направляющие* на панели инструментов

Если у *View* есть двусторонняя вертикальная привязка и значение высоты установлено в *match_constraints* (0 dp), то *View* растянется по высоте между объектами привязки. При использовании *aspect ratio* можно настроить элемент так, чтобы высота не растягивалась, а зависила от ширины *View*. Для *View* с двусторонней вертикальной привязкой надо выставить значение высоты в 0 dp. При этом *View* растягивается по высоте. Затем необходимо включить режим соотношения сторон, нажав на треугольник (рис. 2.22). Задать соотношение ширины

к высоте, например 3 : 1. То есть высота теперь будет в три раза меньше ширины. С изменением ширины меняется и высота, чтобы соблюдалось установленное соотношение сторон 3 : 1 (рис. 2.22).

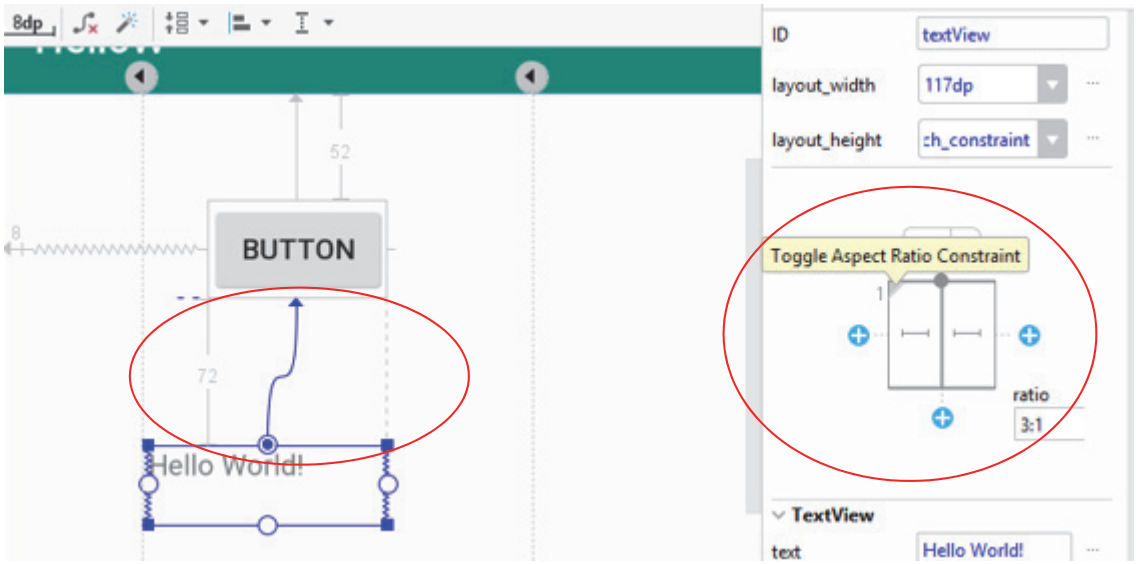


Рис. 2.22. Установка режима соотношения сторон

Цепочка позволит равномерно распределить несколько *View* в имеющемся свободном пространстве. Чтобы создать цепочку, необходимо выделить *View* и центрировать их по горизонтали или вертикали (рис. 2.23).

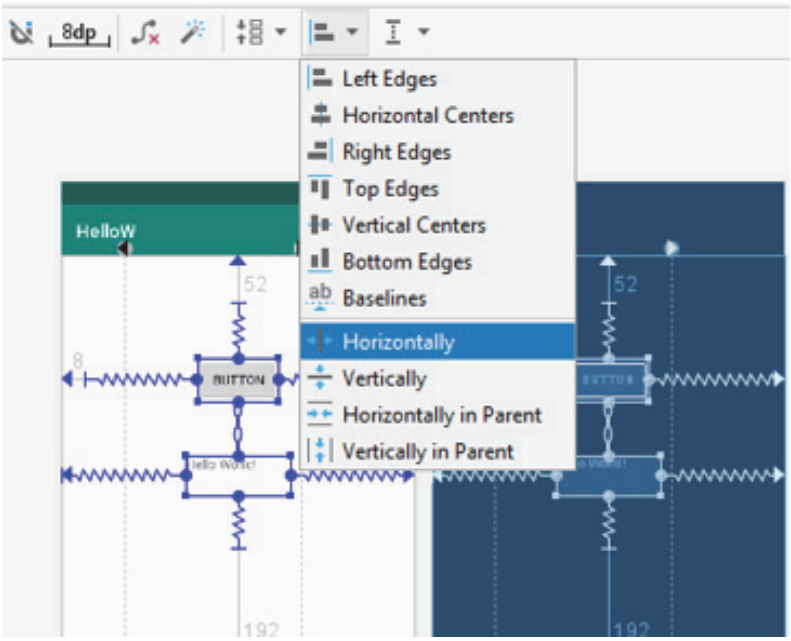


Рис. 2.23. Выбор режима выравнивания

Цепочка может быть в одном из трех режимов.

1. *Spread*: свободное пространство равномерно распределяется между *View* и границами родителя.

2. *Spread_inside*: свободное пространство равномерно распределяется только между *View*. Крайние *View* прижимаются к границам родителя.

3. *Packed*: свободное пространство равномерно распределяется между крайними *View* и границами родителя. Можно использовать *margin*, чтобы сделать отступы между *View*. Режимы цепочки переключаются нажатием на значок цепи (рис. 2.24).

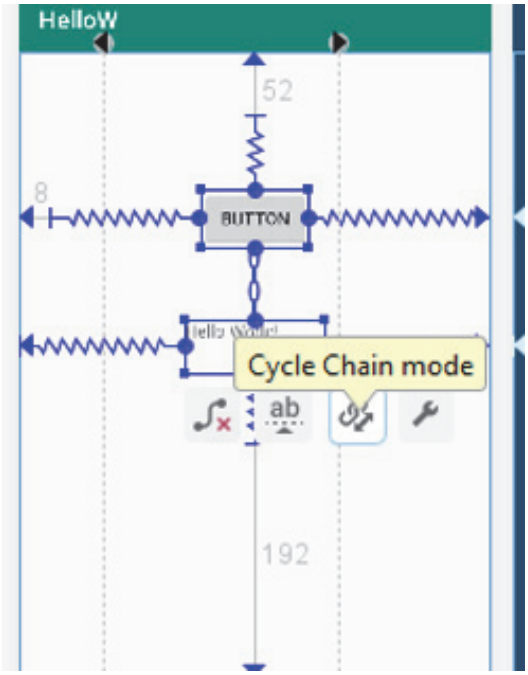


Рис. 2.24. Создание цепочек

В качестве объектов для привязки можно использовать не только границы родителя, но и другие объекты.

Цепочка позволяет указывать для *View* значение веса – *weight*. По умолчанию их нет в основном списке *Properties*. Чтобы увидеть все атрибуты, необходимо нажать на значок с двумя стрелками (рис. 2.25).

Для оптимизации производительности UI необходимо использовать как можно меньшее количество разных *ViewGroup*. Для относительного расположения группы элементов можно выбрать барьеры (рис. 2.26).

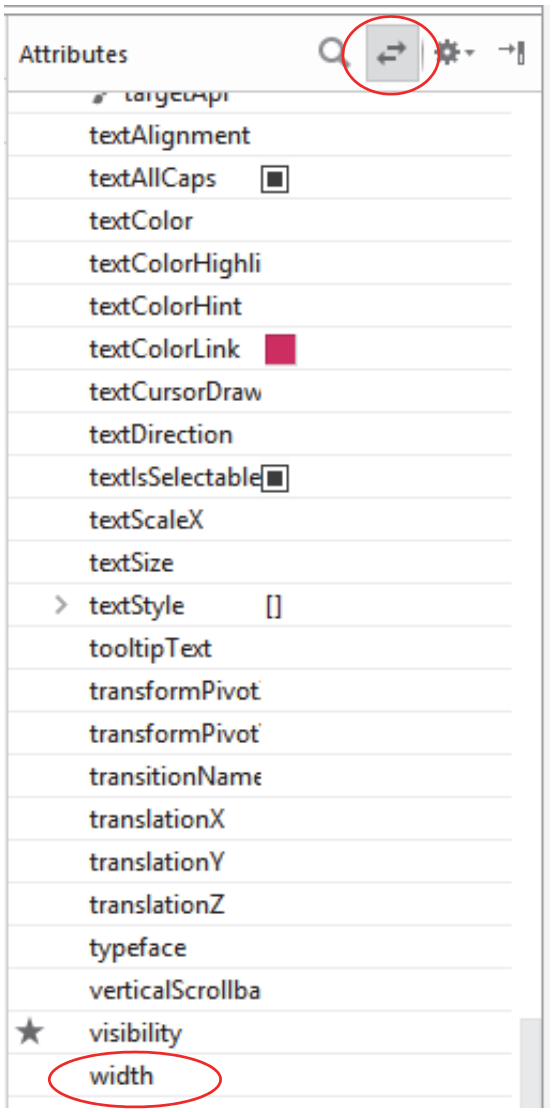


Рис. 2.25. Окно управления атрибутами

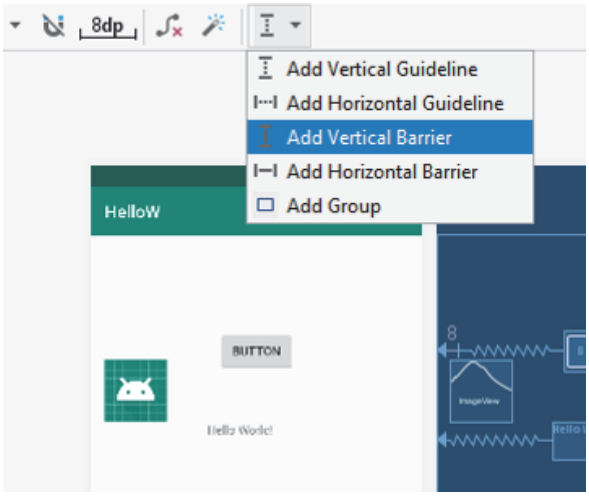


Рис. 2.26. Добавление барьеров

2.3.7. ScrollView

Контейнер *ScrollView* предназначен для создания прокрутки интерфейса, все элементы которого одномоментно не могут поместиться на экране устройства.

2.4. Обращение к View

Чтобы обратиться к элементу *View* из кода, нужен его *ID*. Он прописывается или в *Properties*, или в *layout*-файлах. Для *ID* существует четкий формат: *@+id/name*, где + означает, что это новый ресурс и он должен добавиться в класс *R.java*, если он там еще не существует (рис. 2.27). Идентификатор – целое уникальное число.

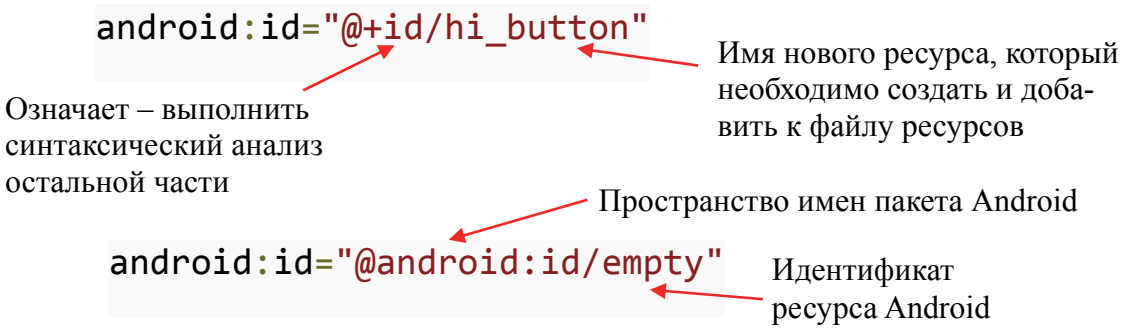


Рис. 2.27. Индентификация *View*

Названия элементов соответствуют названиям классов, а названия атрибутов соответствуют методам. Атрибуты бывают общие, специфические.

Листинг инициализации *Layout:*

```
public class HelloWActivity extends AppCompatActivity {


@Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Устанавливаем в качестве интерфейса файл
        setContentView(R.layout.activity_hello_w);

        // Получаем элемент
        Button buttonHi = (Button) findViewById(R.id.buttonHi);
        // Переустанавливаем у него текст
        buttonHi.setText("ISIT");
}
```

Чтобы обратиться к элементу программно, понадобится метод *findViewById*. Он по *ID* возвращает *View* (см. приведенный выше листинг).

2.5. Ресурсы

2.5.1. Группирование ресурсов

Ресурсы представляют собой файлы разметки, отдельные строки, звуковые файлы, файлы изображений и т. д. Все ресурсы находятся в проекте в папке *res* (рис. 2.28). Для различных типов ресурсов, определенных в проекте, в *res* создаются подпапки (представлены на рис. 2.28).

Если возьмем стандартный проект Android Studio, который создается по умолчанию, то там уже есть несколько папок для различных ресурсов в *res*:

– *animator/* – анимация свойств;
– *anim/* – XML-файлы, определяющие tween-анимацию;
– *color/* – XML-файлы, определяющие список цветов;
– *drawable/* – графические файлы (*.png*, *.jpg*, *.gif*);
– *mipmap/* – графические файлы, используемые для иконок приложения под различные разрешения экранов;
– *layout/* – макеты;
– *menu/* – меню приложения;
– *raw/* – различные файлы, которые сохраняются в исходном виде;
– *values/* – XML-файлы, которые содержат различные значения, например ресурсы строк;
– *xml/* – произвольные XML-файлы.

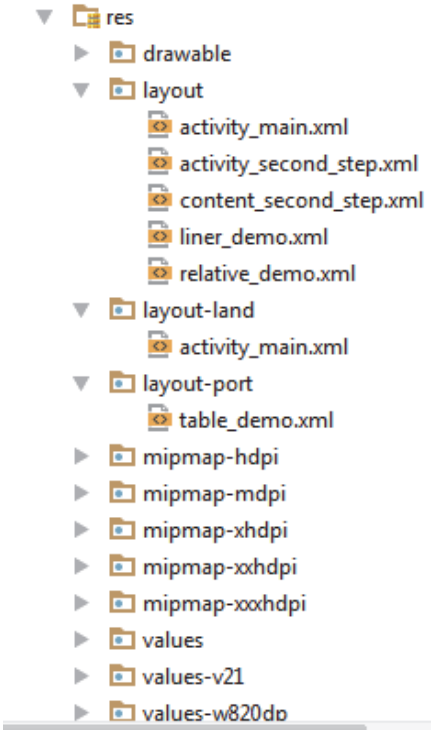


Рис. 2.28. Ресурсы приложения

Макет пользовательского интерфейса по умолчанию сохранен в каталоге *res/layout/*, можно указать другой макет для использования на экране с альбомной ориентацией, сохранив его в каталоге *res/layout-land/*. Android автоматически применяет соответствующие ресурсы, сопоставляя текущую конфигурацию устройства с именами каталогов ресурсов.

Существуют ресурсы по умолчанию и альтернативные ресурсы. Ресурсы по умолчанию должны использоваться независимо от конфигурации устройства или в том случае, когда отсутствуют альтернативные ресурсы, соответствующие текущей конфигурации. Альтернативные ресурсы предназначены для работы с определенными конфигурациями. Чтобы указать, что группа ресурсов предназначена для определенной конфигурации, добавьте соответствующий квалификатор к имени каталога.

2.5.2. Предоставление альтернативных ресурсов и квалификаторы конфигурации

Квалификатор *hdpi* (см. рис. 2.28) указывает, что ресурсы в этом каталоге предназначены для устройств, оснащенных экраном высокой плотности. Идентификатор ресурса всегда одинаков, но Android выбирает версию каждого ресурса, которая оптимально соответствует текущему устройству, сравнивая информацию о его конфигурации с квалификаторами в имени каталога ресурсов.

Язык задается двухбуквенным кодом языка ISO 639-1, к которому можно добавить двухбуквенный код региона ISO 3166-1-alpha-2 (которому предшествует строчная буква *r* для обозначения кода региона). Например en, fr, en-rUS.

Квалификатор *ldrtl* означает «направление макета справа налево». Квалификатор *ldltr* означает «направление макета слева направо» и используется по умолчанию. Эти квалификаторы можно применять к любым ресурсам, таким как макеты, графические элементы или значения.

w<N>dp – указывает минимальную доступную ширину экрана в единицах dp, для которой должен использоваться ресурс, заданный значением <*N*> (например, w720dp). Это значение конфигурации будет изменяться в соответствии с текущей фактической шириной при изменении альбомной/книжной ориентации. Когда приложение предоставляет несколько каталогов ресурсов с разными значениями этой конфигурации, система использует ширину, ближайшую к текущей ширине экрана устройства, но не превышающую ее.

h<N>dp – указывает минимальную доступную высоту экрана в пикселях, для которой должен использоваться ресурс, заданный значением <*N*> (например, h720dp). Это значение конфигурации будет изменяться в соответствии с текущей фактической высотой при изменении альбомной/книжной ориентации.

port – устройство в портретной (вертикальной) ориентации.

land – устройство в книжной (горизонтальной) ориентации. Ориентация может измениться за время работы приложения, если пользователь поворачивает экран.

2.5.3. Строковые ресурсы

По умолчанию для ресурсов строк (хранятся в виде пар «имя – значение строки») применяется файл *strings.xml*, но разработчики могут добавлять дополнительные файлы ресурсов в каталог проекта *res/values*. При этом необходимо соблюдать структуру файла, иметь корневой узел <*resources*> и иметь один или несколько элементов <*string*>:

```
<resources>
    <string name="app_name">ActivityDemo</string>
    <string name="enter">Enter</string>
    <sting name ="login"> Login</sting>
    <string name ="in"> Input name </string>
    <string name="greeting"> Good morning</string>
</resources>
```

Для хранения массива строк используется <*string-array*>:

```
<string-array name="images">
    <item>small</item>
    <item>large</item>
    <item>medium</item>
</string-array>
```

2.5.4. Ресурсы plurals

Plurals предназначены для описания количества элементов (при изменении окончания в зависимости от числительного, которое с ним употребляется). Для задания ресурса используется элемент <*plurals*>, для которого существует атрибут *name*, получающий в качестве значения произвольное название, по которому потом ссылаются на данный ресурс. Сами наборы строк вводятся дочерними элементами <*item*>. Этот элемент имеет атрибут *quantity*, указывающий, когда данная строка используется:

```
?xml version="1.0" encoding="utf-8"?>
<resources>
    <plurals name="student">
      <item quantity="one">%d  студент</item>
      <item quantity="few">%d студентов</item>
      <item quantity="many">%d студентов</item>
    </plurals>
</resources>
```

Использование данного ресурса возможно толко в коде Java.

2.5.5. Ресурсы dimension

Определение размеров должно находиться в каталоге *res/values* в файле с любым произвольным именем. Общий синтаксис определения ресурса следующий:

```
<?xml version="1.0" encoding="utf-8"?>
 <resources>
          <dimen name="имя_ресурса">используемый_размер</dimen>
 </resources>
```

Как и другие ресурсы, *dimension* определяется в корневом элементе *<resources>*. Тег *<dimen>* обозначает ресурс и в качестве значения принимает некоторое значение размера в одной из принятых единиц измерения. Например:

```
<dimen name="activity_horizontal_margin">16dp</dimen>
<dimen name="activity_vertical_margin">16dp</dimen>
<dimen name="text_size">16sp</dimen>
```

Здесь определены два ресурса для отступов *activity_horizontal_margin* и *activity_vertical_margin* и ресурс *text_size* (высота шрифта 16 sp). Названия ресурсов могут быть произвольными.

К ресурсу можно обратиться следующим образом:

```
int leftPadding = (int)getResources()
                    .getDimension(R.dimen.activity_horizontal_margin);
int topPadding = (int)getResources()
                   .getDimension(R.dimen.activity_vertical_margin);
```

Для получения ресурса в XML после "*@dimen/*" указывается имя ресурса:

```
<TextView
    android:textSize="@dimen/text_size"
    android:layout_width="wrap_content"/>
```

2.5.6. Ресурсы color

Ресурсы цветов (*color*) должны храниться в файле по пути *res/values* и также, как и ресурсы строк, заключены в тег <*resources*>. По умолчанию при создании самого простого проекта в папку *res/values* добавляется файл *colors.xml*.

Цвет определяется с помощью элемента <*color*>. Атрибут *name* устанавливает название цвета, которое будет использоваться в приложении, а шестнадцатеричное число – значение цвета:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#3F51B5</color>
    <color name="colorPrimaryDark">#303F9F</color>
    <color name="colorAccent">#FF4081</color>
</resources>
```

Для задания цветовых ресурсов можно использовать следующие форматы: #RGB (#F00 – 12-битное значение); #ARGB (#8F00 – 12-битное значение с добавлением альфа-канала); #RRGGBB (#FF00FF – 24-битное значение); #AARRGGBB (#80FF00FF – 24-битное значение с добавлением альфа-канала).

Для получения цвета применяется метод *ContextCompat.getColor*(), где в качестве первого параметра принимается текущий объект *activity*, а вторым параметром является идентификатор цветового ресурса:

```
int textColor = ContextCompat.getColor(this, R.color.colorPrimary);
```

2.5.7. Доступ к ресурсам

Когда происходит компиляция проекта сведения обо всех ресурсах добавляются *R.java*, который можно найти в проекте по пути *app/build/generated/not_namespaced_r_class_sources/debug/r/[пакет_приложения]* (рис. 2.29).

Существует два способа доступа к ресурсам: в файле исходного кода: *тип_ресурса.имя* (например, *R.string.hello*) и в файле XML: *тип_ресурса/имя* (*@string/hello*).

Для получения ресурсов в классе активности используется метод *getResources*(), который возвращает объект типа *android.content.res.Resources*. Чтобы получить ресурс, надо у объекта *Resources* вызвать один из методов:

– *getString*() – возвращает строку из файла *strings.xml* по числовому идентификатору;

– *getDimension*() – возвращает числовое значение из ресурса *dimen*;

– *getDrawable*() – возвращает графический файл;

– *getBoolean*() – возвращает значение *boolean* и т. д.

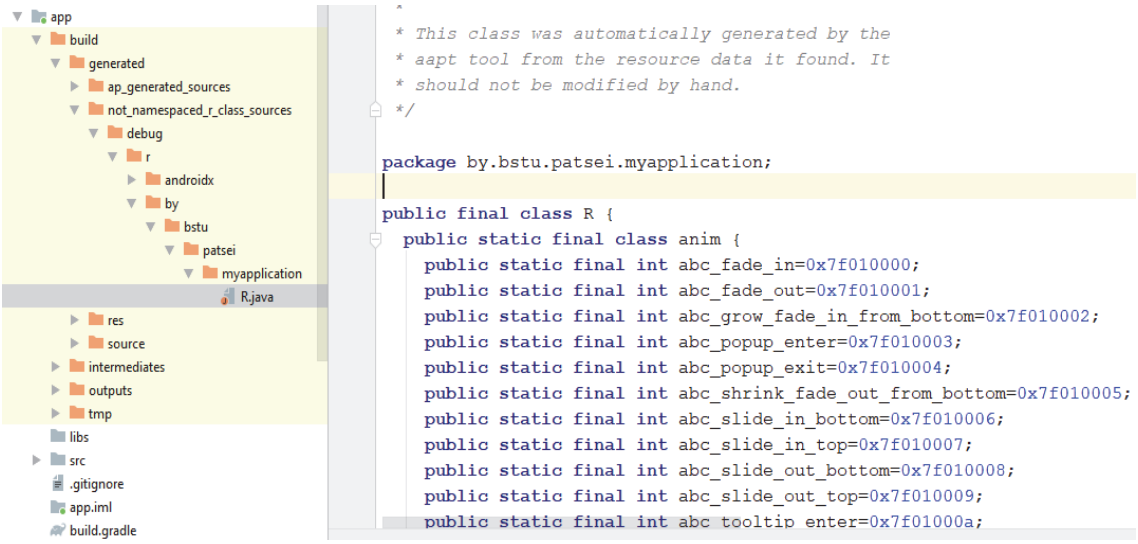


Рис. 2.29. Файл *R.java*

2.5.8. Добавление layout-файла для смены ориентации экрана

По умолчанию *layout*-файл настраивается под вертикальную ориентацию экрана. Если интерфейс насыщен, то для удобства необходим еще один *layout*-файл, который был бы ориентирован под горизонтальную ориентацию и возможно вывел бы элементы по-другому.

Для этого необходимо в папке *res*/*layout* создать новый *layout resource file*. Название файла указывается такое же. Затем добавляется спецификатор, который даст приложению понять, что этот *layout*-файл надо использовать в горизонтальной ориентации. В списке спецификаторов слева снизу находим *Orientation* и включаем использование спецификатора ориентации (рис. 2.30).

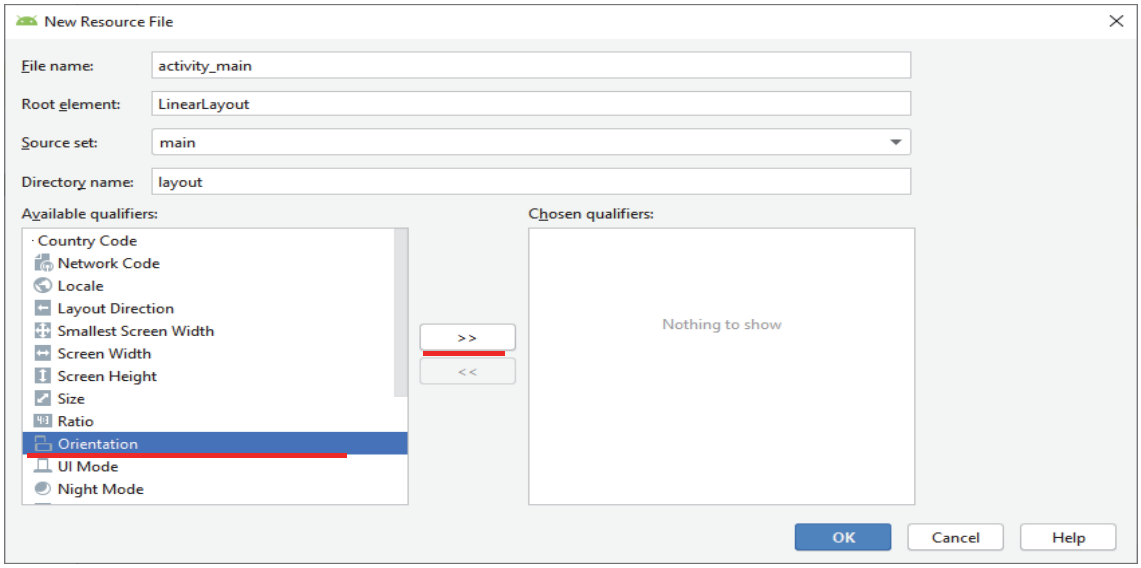


Рис. 2.30. Окно добавления *layout*-файла

Обратите внимание, значение поля *Directory name* изменилось (рис. 2.31). Новый *layout*-файл будет создан в папке *res/layout-land*, а не *res/layout*, как обычно.

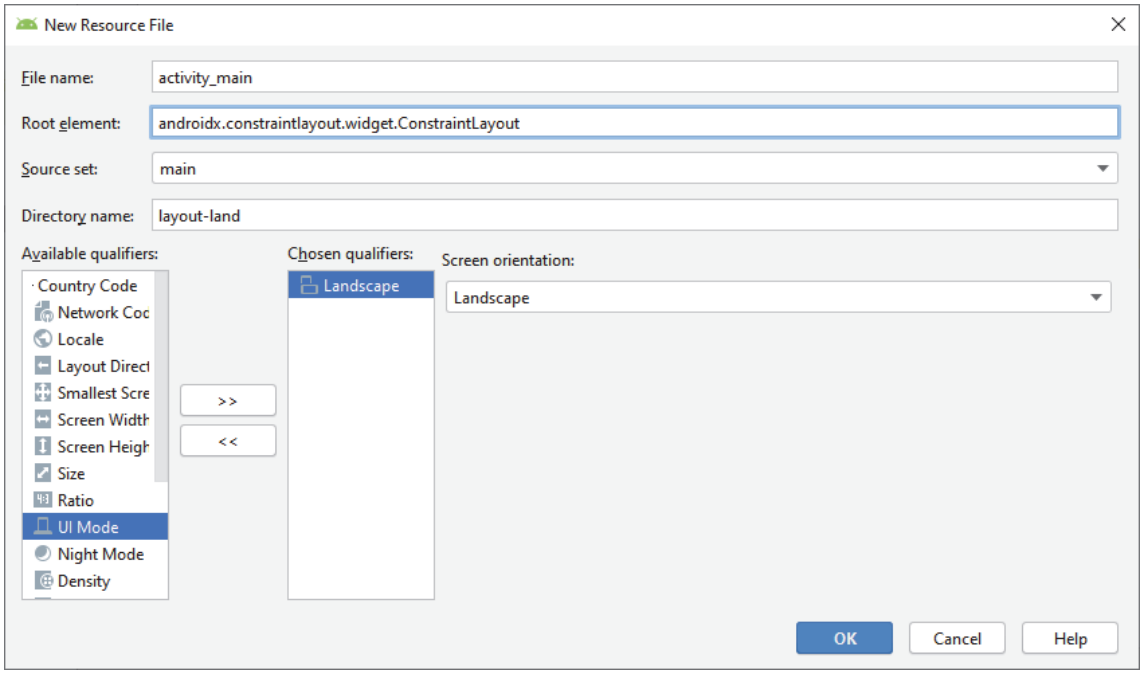


Рис. 2.31. Добавление *layout*-файла для горизонтальной ориентации экрана

Посмотрим на структуру папок проекта (рис. 2.32). Теперь там два файла *activity_main*: обычный и *land*. Каждый из файлов можно менять независимо и соответственно получать разное представление.

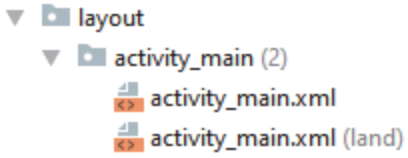


Рис. 2.32. Структура папок проекта

Активность читает *layout*-файл, который указан в методе *setContentView* и отображает его содержимое. При этом учитывается ориентация устройства, и в случае горизонтальной ориентации берется файл из папки *res/layout-land* (если он существует).

Можно управлять сменой шаблона ориентации вручную, если воспользоваться методами, предоставляемыми классом *Configuration*. Также можно запретить смену ориентации экрана для своей активности с помощью атрибута *android:screenOrientation*. Если ограничить приложение одной ориентацией, то шаблон должен располагаться в папке *res/layout*.

2.6. Обработка нажатий View

Пусть есть следующий макет:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:an-
droid="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/textView"
        android:layout_width="114dp"
        android:layout_height="44dp"
        android:layout_marginTop="144dp"
        android:text="@string/helloW"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.474"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="244dp"
        android:text="@string/by"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.197"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_chainStyle="packed" />

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="56dp"
        android:layout_marginTop="36dp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:srcCompat="@mipmap/ic_launcher_round" />

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

В файле *MainActivity.java* описание объектов лучше вынести за пределы метода *onCreate* для того, чтобы можно было обращаться к ним из любого метода:

```
import android.os.Bundle;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    TextView txtView;
    Button btnBy;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
...
```

В методе *onCreate* эти объекты заполняются методом *findViewById*. В методе *onCreate* класса *MainActivity* надо написать следующий код:

```
txtView = (TextView) findViewById(R.id.textView);
btnBy =(Button) findViewById(R.id.button);
```

После того как установлен *layout* и захвачен *View* на экране, с объектами можно работать. Например, можно научить кнопку реагировать на нажатие. Для этого у кнопки есть метод *setOnClickListener*:

```
import android.view.View;

...
        // создаем обработчик нажатия
        // присваеваем обработчик кнопке By
        btnBy.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // меняем текст в TextView
                txtView.setText("The button was clicked");
            }
        });
```

Можно установить обработчик по-другому:

```
btnBy.setOnClickListener((View v) -> {
          txtView.setText("The button was clicked");
});
```

Откомпилируем код. Если произошла ошибка при запуске кода, то можно перейти на вкладку *Logcat* и прочитать информацию об ошибке (рис. 2.33).

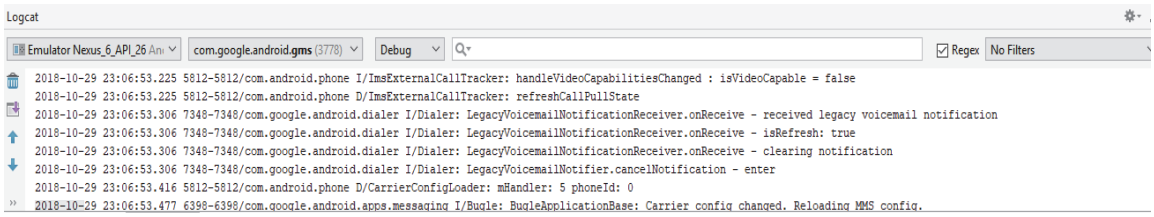


Рис. 2.33. Окно *Logcat*

Если ошибок нет, то после запуска приложения и нажатия на кнопку на экране появится сообщение (рис. 2.34).

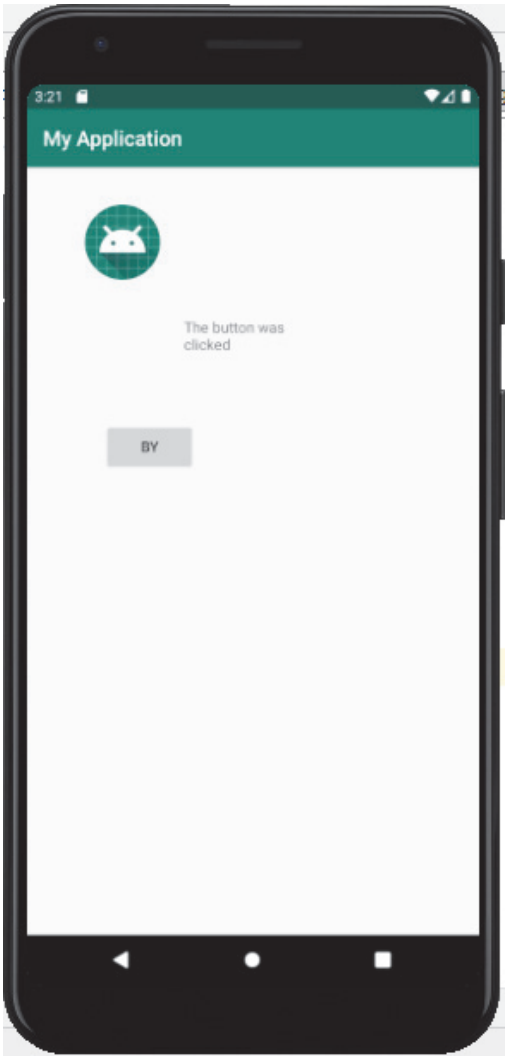


Рис. 2.34. Результат
выполнения приложения

Сама кнопка обрабатывать нажатия не умеет, ей нужен обработчик (его также называют слушателем *listener*), который присваивается с помощью метода *setOnClickListener*. Когда на кнопку нажимают, обработчик реагирует и выполняет код из метода *onClick*.

Затем добавляется еще одна кнопка (рис. 2.35), то есть на экране появятся *TextView* с текстом и две кнопки. Необходимо сделать так, чтобы по нажатию кнопки менялось содержимое *TextView*. По нажатию кнопки *BY* нужно ввести текст «By FIT», по нажатию *Hello* – «Hello FIT». Делается это с помощью одного обработчика, который будет обрабатывать нажатия для обеих кнопок.

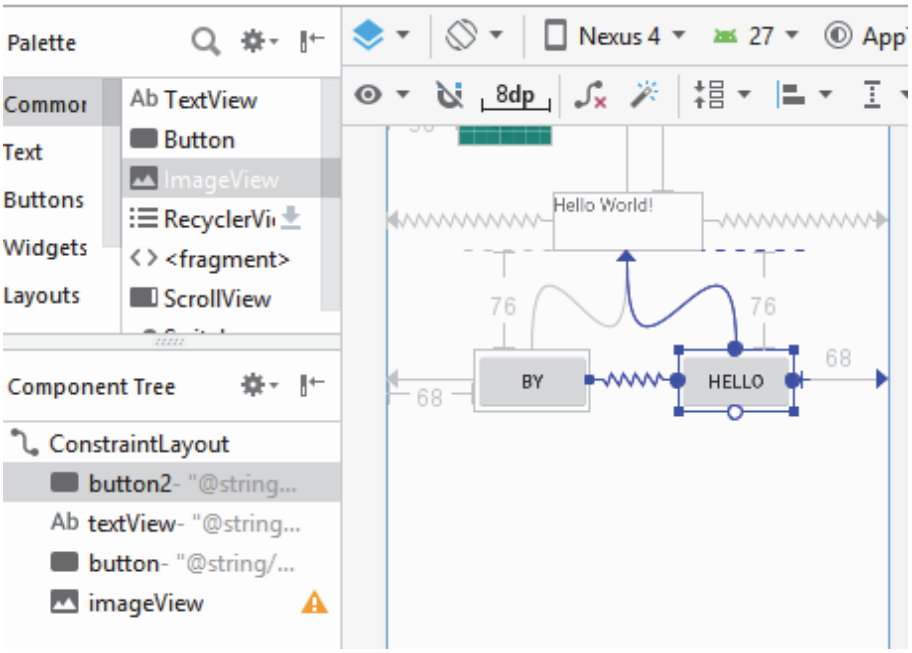


Рис. 2.35. Добавление новой кнопки в активность

Для реализации необходимо выполнить следующие шаги: создать обработчик; заполнить метод *onClick*; присвоить обработчик кнопке. Код переписывается следующим образом:

```
import android.widget.Button;
import android.widget.TextView;
import android.view.View;

public class MainActivity extends AppCompatActivity {

    public final String TAG = "HelloW";
    TextView txtView;
    Button btnBy;
    Button btnHello;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
// Находим View-элементы
        txtView = (TextView) findViewById(R.id.textView);
        btnBy = (Button) findViewById(R.id.button);
        btnHello = (Button) findViewById(R.id.button2);

// Создаем обработчик нажатия
// Присваиваем обработчик кнопке BY
        View.OnClickListener onCLBtn = new View.OnClickListener() {
            // Создаем обработчик
            @Override
            public void onClick(View v) {
                switch (v.getId()) {
                    case R.id.button:
```

```
                    // Кнопка BY
                    txtView.setText("BY FIT");
                    break;
                case R.id.button2:
                    // Кнопка HELLO
                    txtView.setText("Hello FIT");
                    break;
            }
        }
    };
    // Назначение обработчика
    btnBy.setOnClickListener(onCLBtn);
    btnHello.setOnClickListener(onCLBtn);
  }
}
```

В примере использовался обработчик с реализацией метода *onClick*. В параметрах ему подается объект класса *View*. Это объект, на котором произошло нажатие и вызов обработчика. В данном случае это будет либо кнопка *BY,* либо *HELLO*. Поэтому надо узнать *ID* элемента *View* и сравнить его с *R.id.button* и *R.id.button2*, чтобы определить, какая кнопка была нажата. Чтобы получить *ID*, используется метод *getId*.

После создания обработчика его присвоили обеим кнопкам. Результат представлен на рис. 2.36.

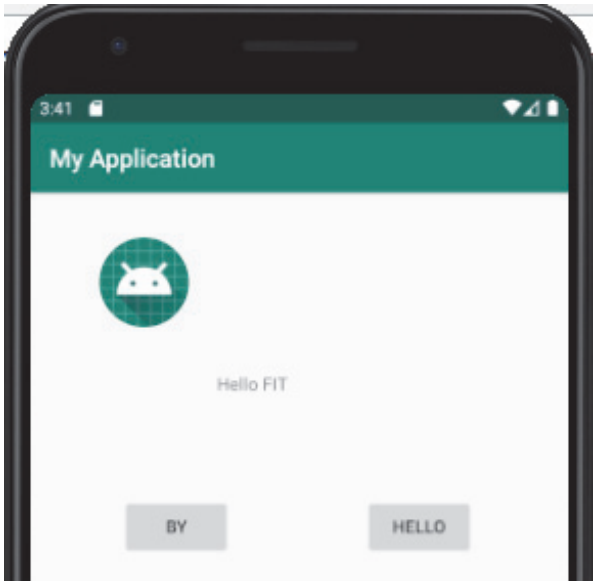


Рис. 2.36. Результат выполнения приложения
с двумя кнопками

Есть еще один способ создания обработчика, который не потребует создания объектов. Можно использовать уже созданный объект активности. Для этого нужно указать, что класс активности реализует интерфейс *View.OnClickListener*, и определить метод *onCreate*:

```
public class MainActivity extends AppCompatActivity  implements
View.OnClickListener{
...
@Override
    public void onClick(View v) {
    }
}
```

Есть еще один способ. В *layout*-файле при определении кнопки используется атрибут *onClick*. В нем указывается имя метода активности. Этот метод срабатывает при нажатии на кнопку:

```
<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="76dp"
    android:layout_marginEnd="68dp"
    android:text="@string/hello"
    android:onClick="onClick"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.927"
    app:layout_constraintStart_toEndOf="@+id/button"
    app:layout_constraintTop_toBottomOf="@+id/textView"
    app:layout_constraintVertical_chainStyle="packed" />
```

Далее добавляется этот метод в *activity* (*MainActivity.java*). Требования к методу: модификатор *public*, возвращаемое значение *void* и параметр типа *View*:

```
public void onClick (View v) {
    // Действия при нажатии на кнопку
}
```

2.7. Вывод Log-сообщений

Когда тестируется работа приложения, полезно видеть логи работы. Они отображаются в окне *Logcat*. Логи имеют разные уровни важности: *ERROR*, *WARN*, *INFO*, *DEBUG*, *VERBOSE* (в порядке убывания критичности). Писать логи можно с помощью класса *Log* и его методов *Log*.v(), *Log*.d(), *Log*.i(), *Log*.w() и *Log*.e(). Названия методов соответствуют уровню логов, которые они пишут, параметры методов – тег и текст сообщения. Тег – это метка, чтобы позволяющая быстро находить нужное сообщение в системных логах.

Затем следует изменить код активности и вывести лог с помощью метода *Log*.d:

```
public class MainActivity extends AppCompatActivity {

public  final String TAG = "HelloW";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
         Log.d(TAG, "On creat");
    }
}
```

После запуска приложения можно изучить сообщения в *Logcat* (рис. 2.37). Когда логов много, их можно отфильтровать по типу сообщения (рис. 2.38, 2.39).

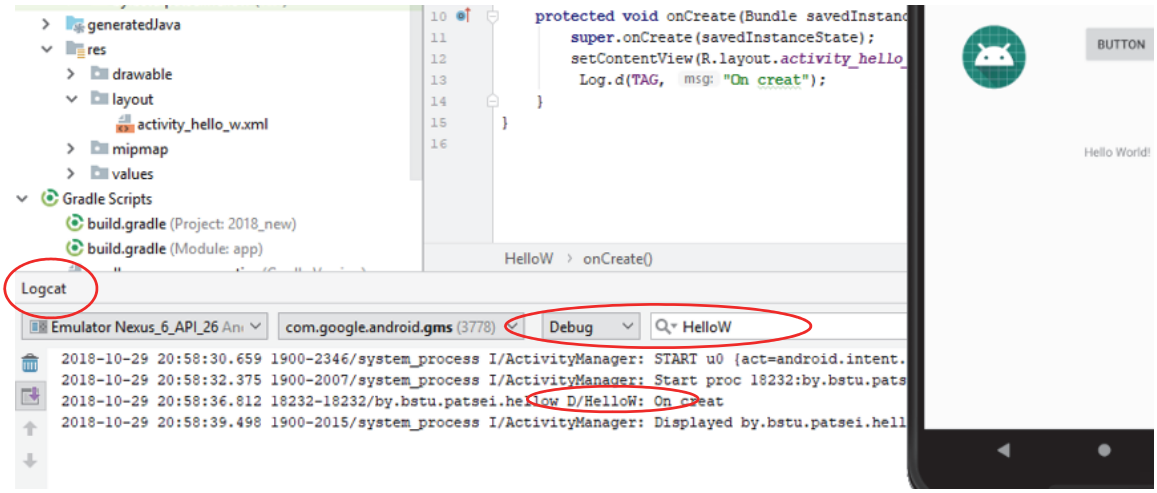


Рис. 2.37. Вывод *Log*-сообщений

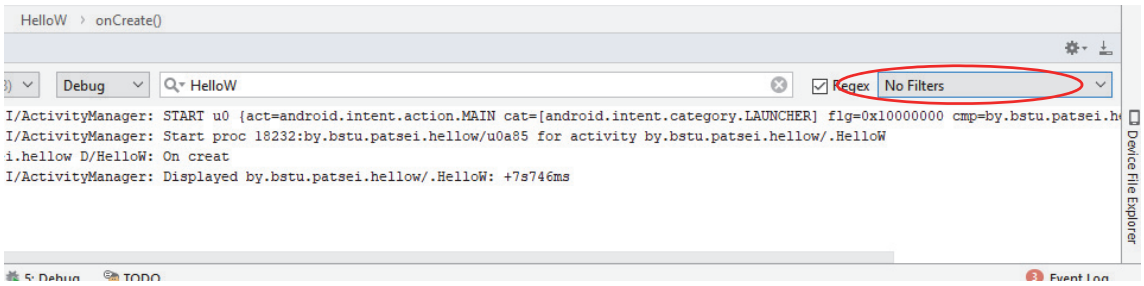


Рис. 2.38. Фильтрация *Log*-сообщений

Можно изменить код *HelloW.java* и добавить *Info*-логи:

```
View.OnClickListener onCLBtn = new View.OnClickListener() {
    // Создается обработчик
    @Override
    public void onClick(View v) {

        switch (v.getId()) {
```

```
            case R.id.button:
                // Кнопка BY
                txtView.setText("BY FIT");
                Log.i ("ActivityHelloW", "button By clicked");
                break;

            case R.id.button2:
                // Кнопка HELLO
                txtView.setText("Hello FIT");
                Log.i ("ActivityHelloW", "button Hello clicked");
                break;
        }
    }
};
```

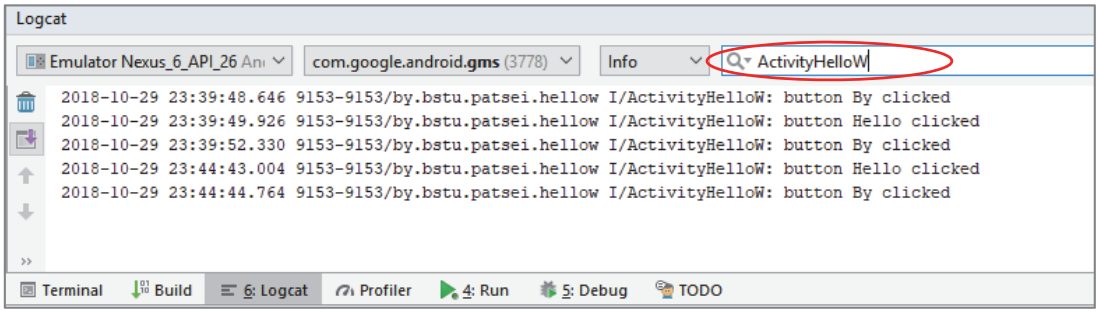


Рис. 2.39. Фильтрация сообщений

2.8. Всплывающие окна Toast

Для создания простых уведомлений в Android используется класс *Toast*. Фактически *Toast* представляет всплывающее окно с текстом, которое отображается в течение некоторого времени.

Объект *Toast* нельзя создать в коде разметки XML. Для его создания применяется метод *Toast.makeText*(), в который передается три параметра: *context* – текущий контекст (текущий объект активности), *text* – отображаемый текст и *duration* – время отображения окна (*Toast.LENGTH*_LONG – 3500 мс, *Toast.LENGTH*_SHORT – 2000 мс):

```
Toast.makeText(getApplicationContext(),"HelloW",Toast.LENGTH_LONG)
     .show();
```

При редактировании метода *onClick* следует сделать так, чтобы всплывало сообщение о том, какая кнопка была нажата:

```
View.OnClickListener onCLBtn = new View.OnClickListener() {

    // Создаем обработчик
    @Override
```

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

            case R.id.button:
                txtView.setText("BY FIT");
                Log.i ("ActivityHelloW", "button By clicked");
                Toast.makeText(getApplicationContext(), "Button By
clicked", Toast.LENGTH_LONG).show();
                break;

            case R.id.button2:
                txtView.setText("Hello FIT");
                Log.i ("ActivityHelloW", "button Hello clicked");
                Toast.makeText(getApplicationContext(), "Button Hello
clicked", Toast.LENGTH_LONG).show();
                break;
        }
    }
};
```

Результат выполнения представлен на рис. 2.40.

Рис. 2.40. Результат выполнения приложения
с всплывающими сообщениями

2.9. Элементы управления

2.9.1. TextView

TextView предназначен для вывода текста на экран без возможности его редактирования. Его основные атрибуты:

– *android*:*text* – устанавливает текст элемента;
– *android*:*textSize* – устанавливает высоту текста;
– *android*:*background* – задает фоновый цвет элемента в виде цвета в шестнадцатеричной записи или в виде цветового ресурса;
– *android*:*textColor* – задает цвет текста;
– *android*:*textAllCaps* – при значении *true* делает все символы в тексте заглавными;
– *android*:*textDirection* – устанавливает направление текста. По умолчанию используется направление слева направо;
– *android*:*textAlignment* – задает выравнивание текста;
– *android*:*fontFamily* – устанавливает тип шрифта.

Иногда необходимо вывести на экран какую-нибудь ссылку либо телефон, по нажатию на которые выполнялось бы определенное действие. Для этого в *TextView* определен атрибут *android*:*autoLink*, который принимает значения:

– *none* – отключает все ссылки;
– *web* – включает все веб-ссылки;
– *email* – включает ссылки на электронные адреса;
– *phone* – включает ссылки на номера телефонов;
– *map* – включает ссылки на карту;
– *all* – включает все вышеперечисленные ссылки.

2.9.2. EditText

Элемент *EditText* является подклассом класса *TextView*. Он также представляет текстовое поле, но с возможностью ввода и редактирования текста. В *EditText* можно использовать все те же возможности, что и в *TextView*.

Из тех атрибутов, что не рассматривались, следует отметить *android*:*hint*. Он позволяет задать текст, который будет отображаться в качестве подсказки, если элемент *EditText* пуст. Кроме того, можно использовать атрибут *android*:*inputType*, который позволяет задать клавиатуру для ввода:

– *text* – обычная клавиатура для ввода однострочного текста;
– *textMultiLine* – многострочное текстовое поле;
– *textEmailAddress* – обычная клавиатура, на которой присутствует символ @, ориентирована на ввод email;

— *textUri* – обычная клавиатура, на которой присутствует символ /, ориентирована на ввод интернет-адресов;

— *textPassword* – клавиатура для ввода пароля;

— *textCapWords* – первый введенный символ слова при вводе представляет заглавную букву, остальные – строчные;

— *number* – числовая клавиатура;

— *phone* – клавиатура в стиле обычного телефона;

— *date* – клавиатура для ввода даты;

— *time* – клавиатура для ввода времени;

— *datetime* – клавиатура для ввода даты и времени.

Элемент *EditText* определяется так (результат представлен на рис. 2.41):

```
<EditText
    android:layout_marginTop="16dp"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:hint="Введите сообщение"
    android:inputType="textMultiLine"
    android:gravity="top"  />
```

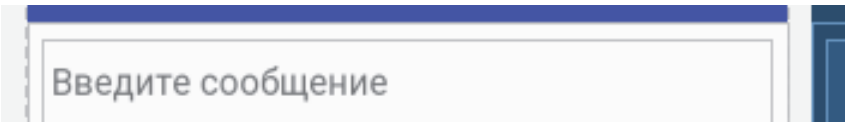


Рис. 2.41. Вывод элемента *EditText*

2.9.3. Button

Одним из часто используемых элементов являются кнопки, которые представлены классом *android.widget.Button*. Ключевой особенностью кнопок является возможность взаимодействия с пользователем через нажатия.

Ниже представлены некоторые ключевые атрибуты, которые можно присвоить кнопкам:

— *text* – задает текст на кнопке;

— *textColor* – задает цвет текста;

— *background* – задает фоновый цвет;

— *textAllCaps* – при значении *true* устанавливает текст в верхнем регистре;

— *onClick* – определяет обработчик нажатия кнопки.

2.9.4. Checkboxes

Элементы *Checkbox* представляют собой флажки, которые могут находиться в отмеченном и неотмеченном состоянии. Флажки позволяют производить множественный выбор. С помощью слушателя *OnCheckedChangeListener* можно отслеживать изменения флажка.

2.9.5. RadioButton

Схожую с флажками функциональность предоставляют переключатели, которые представлены классом *RadioButton*. Чтобы создать список переключателей для выбора, вначале надо создать объект *RadioGroup*, который будет включать в себя все переключатели.

2.9.6. ToggleButton

Атрибуты *android*:*textOn* и *android*:*textOff* задают текст кнопки в отмеченном и неотмеченном состоянии соответственно. И так же, как и для других кнопок, можно обработать нажатие на элемент с помощью события *onClick*:

```
<ToggleButton
    android:id="@+id/toggleButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:textOn="ВКЛ"
    android:textOff="ВЫКЛ"/>
```

3. ПОНЯТИЕ ACTIVITY И ЖИЗНЕННЫЙ ЦИКЛ ACTIVITY

3.1. Понятие активности (Activity)

Каждая *активность* (*Activity*) – это экран (по аналогии с формой), который приложение может показывать пользователям. Чем сложнее создаваемое приложение, тем больше экранов (*Activity*) потребуется. При создании приложения потребуется как минимум начальный (главный) экран. При необходимости этот интерфейс дополняется второстепенными активностями, предназначенными для ввода информации, вывода и предоставления дополнительных возможностей. Запуск новой активности или возврат из нее приводит к «перемещению» между экранами.

Для создания новой активности необходимо наследование от класса *Activity* или *AppCompatActivity*. Внутри реализации класса надо определить пользовательский интерфейс и реализовать требуемый функционал. Базовый каркас для новой активности выглядит следующим образом:

```
public class MainActivity extends AppCompatActivity  {

/** Вызывается при создании Активности */
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState); }
}
```

Базовый класс *Activity* (*AppCompatActivity*) представляет собой пустой экран, который не особенно полезен, поэтому первое, что нужно сделать, это создать пользовательский интерфейс с помощью представлений (*View*) и разметки (*Layout*).

Для использования активности в приложении ее необходимо зарегестрировать в файле манифеста путем добавления элемента <*activity*> внутри узла <*application*>, в противном случае ее невозможно будет использовать:

```
<activity
    android:name=".MainActivity"
    android:label="@string/app_name"
```

```
    android:theme="@style/AppTheme.NoActionBar">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

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

В тег <*activity*> можно добавить элементы <*intent-filter*> для указания *намерений* (*Intent*), которые активность будет отслеживать.

3.2. Жизненный цикл Activity

Приложения Android не могут контролировать свой жизненный цикл. ОС сама управляет всеми процессами и, как следствие, активностями внутри них. При этом состояние активности помогает ОС определить приоритет родительской активности. А приоритет приложения влияет на то, с какой вероятностью его работа (и работа дочерних активностей) будет прервана системой.

3.2.1. Стек Activity

Состояние каждой активности определяется ее позицией в стеке активностей, запущенных в данный момент. При запуске новой *Activity* представляемый ею экран помещается на вершину стека. Если пользователь нажимает кнопку *Назад* или эта активность закрывается каким-то другим образом, на вершину стека перемещается (и становится активной) нижележащая активность (рис. 3.1).

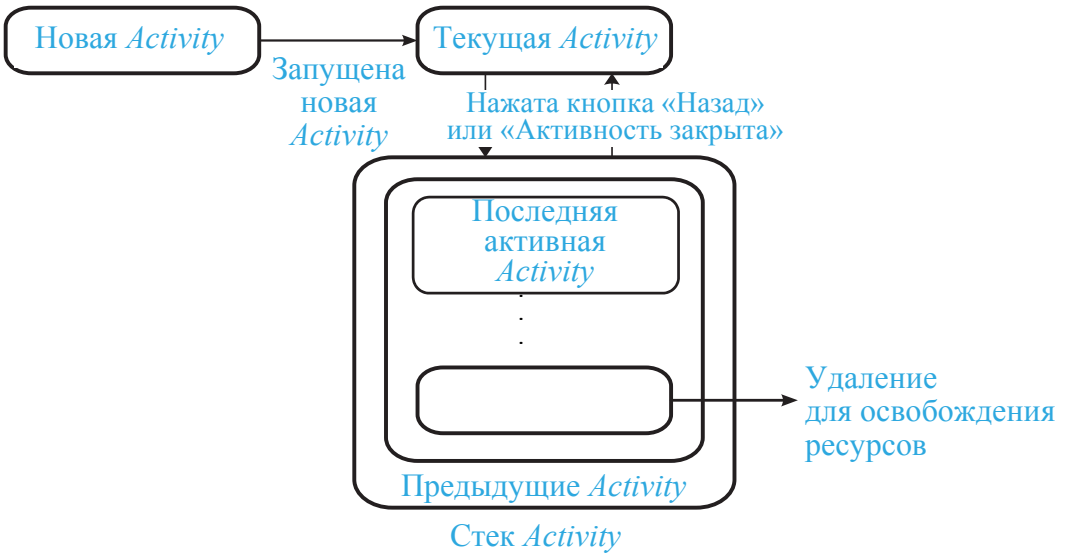


Рис. 3.1. Стек активностей

На приоритет приложения влияет его самая приоритетная активность. Когда диспетчер памяти ОС решает, какую программу закрыть для освобождения ресурсов, он учитывает информацию в стеке для определения приоритета приложения.

3.2.2. Состояния Activity

Activity могут находиться в одном из четырех возможных состояний.

Active (активное). *Activity* находится на переднем плане (на вершине стека) и имеет возможность взаимодействовать с пользователем. Android будет пытаться сохранить ее работоспособность любой ценой, при необходимости прерывая работу других *Activity*, находящихся на более низких позициях в стеке. При выходе на передний план другой *Activity* работа данной Активности будет *приостановлена* или *остановлена*.

Paused (приостановленное). В этом состоянии *Activity* может быть видна на экране, но не может взаимодействовать с пользователем: в этот момент она приостановлена. Это случается, когда на переднем плане находятся полупрозрачные или плавающие (например, диалоговые) окна. Работа приостановленной активности может быть прекращена, если ОС необходимо выделить ресурсы активности переднего плана. Если активность полностью исчезает с экрана, она *останавливается*.

Stopped (остановленное). *Activity* невидима, она находится в памяти, сохраняя информацию о своем состоянии. Такая активность становится кандидатом на преждевременное закрытие, если системе потребуется память. При остановке активности разработчику важно сохранить данные и текущее состояние пользовательского интерфейса (состояние полей ввода, позицию курсора и т. д.). Если активность завершает свою работу или закрывается, она становится *неактивной*.

Inactive (неактивное). Когда работа *Activity* завершена, она находится в неактивном состоянии. Такие активности удаляются из стека и должны быть перезапущены, чтобы их можно было использовать.

Изменение состояния приложения – недетерминированный процесс и управляется исключительно менеджером памяти Android. При необходимости Android вначале закрывает приложения, содержащие *неактивные Activity*, затем *остановленные* и в крайнем случае *приостановленные*.

Для обеспечения полноценного интерфейса приложения изменения его состояния должны быть незаметными для пользователя. При остановке или приостановке работы активности необходимо обеспечить сохранение ее состояния, которое можно восстановить при выходе активности на передний план. Для этого в классе *Activity* имеются обработчики событий, переопределение которых позволяет разработчику отслеживать изменение состояний активностей.

3.2.3. Отслеживание изменения состояний Activity

Обработчики событий класса *Activity* позволяют отслеживать изменение состояний соответствующего объекта активности во время всего жизненного цикла (рис. 3.2):

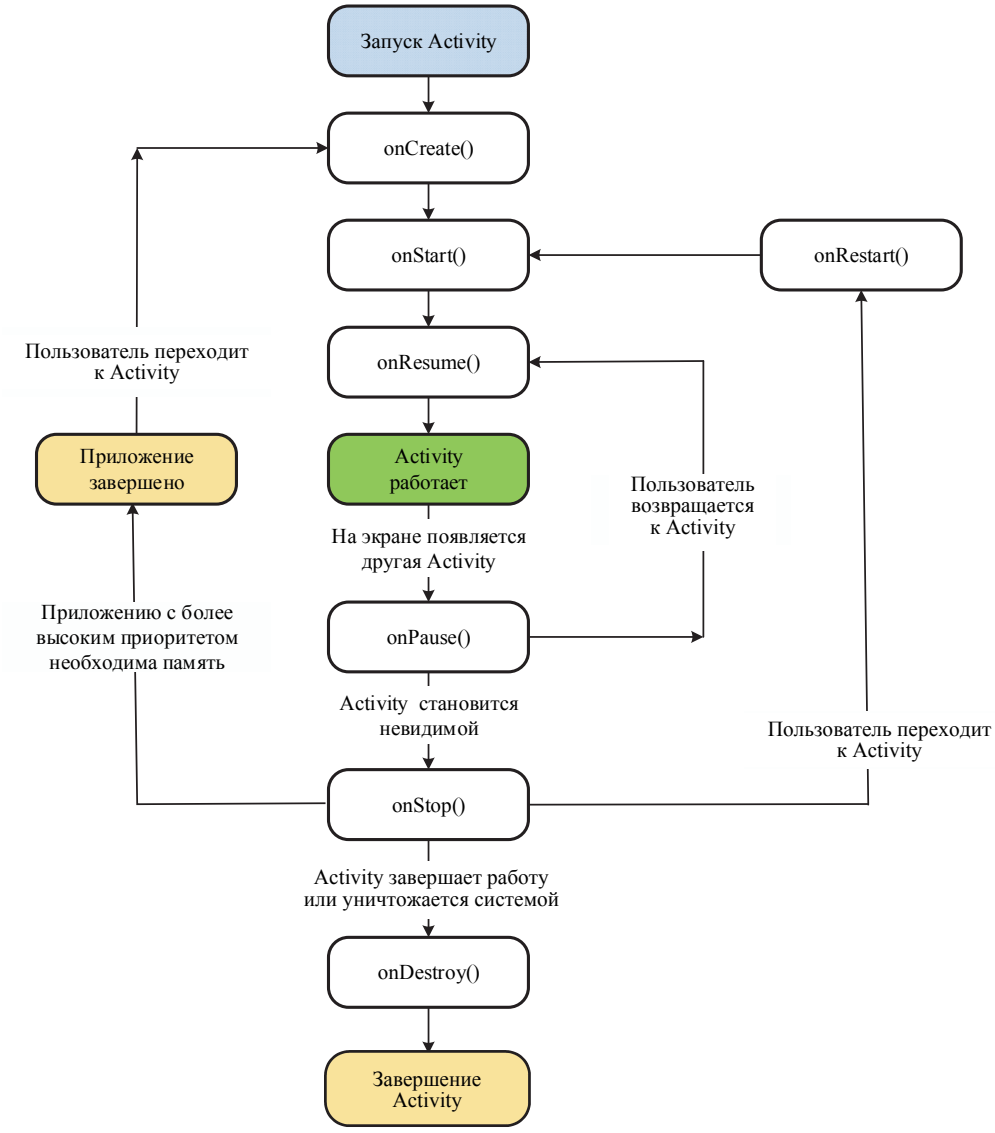


Рис. 3.2. Обработчики событий *Activity*

```
protected void onCreate(Bundle saveInstanceState);
protected void onStart();
protected void onRestoreInstanceState
                    (Bundle saveInstanceState);
protected void onRestart();
protected void onResume();
protected void onPause();
protected void onSaveInstanceState(Bundle saveInstanceState);
protected void onStop();
protected void onDestroy();
```

Рассмотрим методы обратного вызова *Activity*.

***onCreate*(*Bundle saveInstanceState*).** Это первый метод, с которого начинается выполнение активности (рис. 3.3). В нем производится первоначальная настройка активности. В частности, создаются объекты визуального интерфейса. В качестве параметров метод получает объект *Bundle*, который содержит прежнее состояние активности, если оно было сохранено (если создается заново, то *null*). Этот метод необходимо обязательно реализовать, поскольку система вызывает его при создании *Activity*. Здесь нужно инициализировать ключевые компоненты и вызвать *setContentView*() для определения макета пользовательского интерфейса.

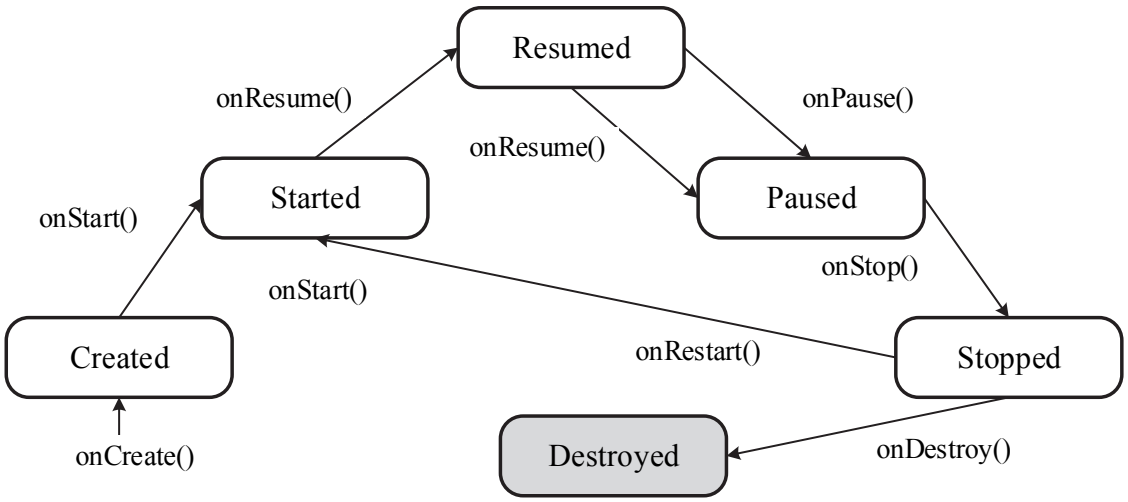


Рис. 3.3. Состояния *Activity* и переходы между состояниями

***onStart*().** В методе *onStart*() осуществляется подготовка к выводу активности на экран устройства. Как правило, этот метод не требует переопределения и всю работу производит встроенный код. После завершения работы метода активность отображается на экране, вызывается метод *onResume*, а активность переходит в состояние *Resumed*.

***onRestoreInstanceState*().** После завершения метода *onStart*() вызывается метод *onRestoreInstanceState*(), который призван восстанавливать сохраненное состояние из объекта *Bundle*. Этот метод вызывается только тогда, когда *Bundle* не равен *null* и содержит ранее сохраненное состояние. Так, при первом запуске приложения метод *onRestoreInstanceState*() не будет вызываться.

***onResume*().** Вызывается непосредственно перед тем, как активность начинает взаимодействие с пользователем. На этом этапе активность находится в самом верху стека активностей и в нее поступают данные пользователя.

***onPause*().** Если пользователь решит перейти к другой активности, то система вызывает метод *onPause*. В этом методе можно освобождать используемые ресурсы, приостанавливать процессы, останавливать работу камеры (если она используется) и т. д., чтобы они меньше сказывались на производительности системы. Но на работу данного метода отводится очень мало времени, поэтому не стоит здесь сохранять какие-то данные, особенно если при этом требуется обращение к сети, например, отправка данных или обращение к базе данных.

После выполнения этого метода активность становится невидимой, не отображается на экране, но она все еще активна. Если пользователь решит вернуться к ней, то система снова вызовет метод *onResume*() и активность опять появится на экране (см. рис. 3.3).

Другой вариант работы может возникнуть, если система видит, что для работы активных приложений необходимо больше памяти. В этом случае ОС может сама завершить работу активности, которая невидима и находится в фоне.

***onSaveInstanceState*(*Bundle saveInstanceState*).** Метод *onSaveInstanceState*() вызывается после метода *onPause*(), но до вызова *onStop*(). В *onSaveInstanceState* производится сохранение состояния приложения в передаваемом в качестве параметра объекте *Bundle*.

Если ОС уничтожает активность в целях восстановления памяти, объект уничтожается, в результате чего системе не удается просто восстановить состояние активности. Вместо этого ОС необходимо повторно создать активность. Но пользователю неизвестно, что система уже ее уничтожила и создала повторно, поэтому он, скорее всего, ожидает, что *Activity* осталась прежней. В этой ситуации можно обеспечить сохранение важной информации о состоянии активности путем реализации дополнительного метода обратного вызова, который позволяет сохранить информацию (см. рис. 3.3).

***onStop*().** Вызывается в случае, когда активность больше не отображается для пользователя. Пользователь может нажать на кнопку *Back* (*Назад*). В этом случае у *Activity* вызывается метод *onStop*().

Система вызывает этот метод в качестве первого признака выхода пользователя из *Activity* (это не всегда означает, что она будет уничтожена).

***onDestroy*().** Завершается работа активности вызовом метода *onDestroy*(), который возникает, если система решит убить активность либо при вызове метода *finish*().

При изменении ориентации экрана ОС завершает активность и затем создает ее заново, вызывая метод *onCreate*(). В большинстве случаев не следует ее явно завершать. Android выполнит такое управление за вас. Вызов методов завершения может отрицательно сказаться на ожидаемом поведении приложения.

3.2.4. Сохранение состояния Activity

Из описания методов обратного вызова активности очевидно, что есть два способа возврата к отображению для пользователя в неизменном состоянии: уничтожение с последующим повторным созданием, когда активность должна восстановить свое ранее сохраненное состояние, или остановка активности и ее последующее восстановление в неизмененном состоянии (рис. 3.4).

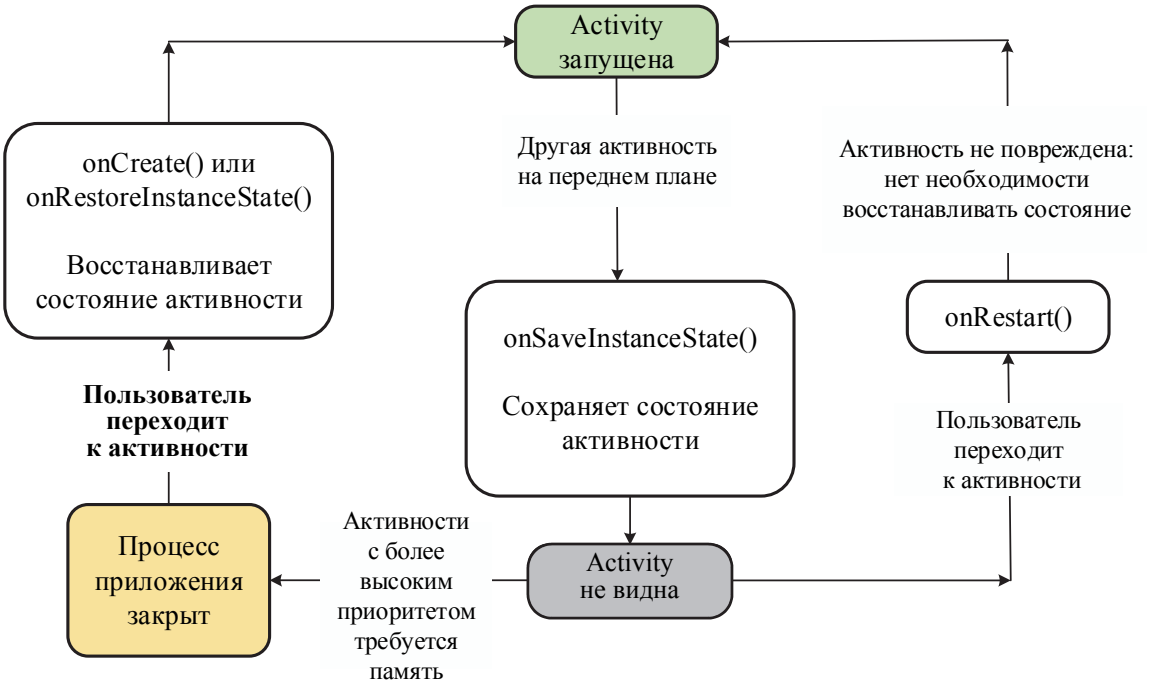


Рис. 3.4. Сохранение состояния *Activity*

Нет никаких гарантий, что метод *onSaveInstanceState*() будет вызван до того, как активность будет уничтожена. Если система вызывает метод *onSaveInstanceState*(), она делает это до вызова метода *onStop*(). Однако даже если не реализовать метод *onSaveInstanceState*(), часть состояния *Activity* восстанавливается реализацией по умолчанию. В частности, реализация по умолчанию вызывает соответствующий метод *onSaveInstanceState*() для каждого объекта *View* в макете, благодаря чему каждое представление может предоставлять ту информацию о себе, которую следует сохранить.

Поскольку вызов метода *onSaveInstanceState*() не гарантируется, следует использовать его только для записи переходного состояния активности (никогда не надо использовать его для хранения постоянных данных). Лучше пользоваться для этого методом *onPause*().

Для того чтобы проверить возможность приложения восстанавливать свое состояние, надо повернуть устройство для изменения ориентации экрана. При изменении ориентации экрана ОС уничтожает и повторно создает *Activity*, чтобы применить альтернативные ресурсы, которые могут быть доступны для новой конфигурации экрана.

Для понимания жизненного цикла активности следует переопределить методы *onPause*, *onStart*, *onRestart* для класса и внести изменения в метод *onCreate*:

```
public class MainActivity extends AppCompatActivity  {

    // Вызывается при создании Активности
    @Override
    public void onCreate(Bundle savedInstanceState)     {
        super.onCreate(savedInstanceState);
    }
// Инициализирует Активность

// Вызывается после завершения метода onCreate
// Используется для восстановления состояния UI
        @Override
        public void onRestoreInstanceState(Bundle savedInstanceState)
{
            super.onRestoreInstanceState(savedInstanceState);
 // Восстанавливаем состояние UI из объекта savedInstanceState
// Данный объект также был передан методу onCreate
        }
// Вызывается перед тем, как Активность снова становится видимой
        @Override
        public void onRestart(){
        super.onRestart();
// Восстанавливаем состояние UI с учетом того,
// что данная Активность уже была видимой
        }
// Вызывается когда Активность стала видимой
        @Override public void onStart(){
            super.onStart();
// Проделываем необходимые действия для
// Активности, видимой на экране
        }
// Должен вызываться в начале видимого состояния.
// На самом деле Android вызывает данный обработчик только
// для Активностей, восстановленных из неактивного состояния
        @Override public void onResume(){
            super.onResume();
// Восстанавливаем приостановленные обновления UI,
// потоки и процессы, «замороженные», когда
// Активность была в неактивном состоянии
        }

// Вызывается перед выходом из активного состояния,
// позволяя сохранить состояние в объекте savedInstanceState
        @Override
        public void onSaveInstanceState(Bundle savedInstanceState) {
```

```
// Объект savedInstanceState будет в последующем
// передан методам onCreate и onRestoreInstanceState
     super.onSaveInstanceState(savedInstanceState);
        }
// Вызывается перед выходом из активного состояния
            @Override
            public void onPause(){
// «Замораживаем» обновления UI, потоки или
// «трудоемкие» процессы, ненужные, когда Активность
// не на переднем плане
             super.onPause();
            }
// Вызывается перед выходом из видимого состояния
            @Override
            public void onStop(){
// «Замораживаем» обновления UI, потоки или
// «трудоемкие» процессы, ненужные, когда Активность
// не на переднем плане.
// Сохраняем все данные и изменения в UI, так как
// процесс может быть в любой момент убит системой
       super.onStop();
            }
// Вызывается перед уничтожением активности
            @Override
            public void onDestroy(){
// Освобождаем все ресурсы, включая работающие потоки,
// соединения с БД и т. д.
                super.onDestroy();
            }
        }
```

Теперь необходимо добавить вывод *Toast*:

```
@Override
    public void onCreate(Bundle savedInstanceState) { super.onCre-
ate(savedInstanceState); setContentView(R.layout.activity_main);
        Toast.makeText(this, "onCreate()", Toast.LENGTH_LONG).show();
    }
    @Override
    protected void onPause() {
        Toast.makeText(this, "onPause()", Toast.LENGTH_LONG).show();
super.onPause();
    }
    @Override
    protected void onRestart() { super.onRestart();
        Toast.makeText(this, "onRestart()", Toast.LENGTH_LONG).show();
    }
    @Override
    protected void onStart() { super.onStart();
        Toast.makeText(this, "onStart()", Toast.LENGTH_LONG).show();
    }
}
```

Можно запустить приложение, посмотреть результат и изменить ориентацию экрана.

4. ПОНЯТИЕ INTENT. ВЗАИМОДЕЙСТВИЕ ACTIVITY

4.1. Добавление *Activity*

Как правило, приложение состоит из нескольких активностей, которые связаны (иногда слабо) друг с другом. Обычно одна из *Activity* в приложении обозначается как «основная» (отображается при первом запуске приложения). В свою очередь, каждая *Activity* может запустить другую *Activity*. При этом предыдущая останавливается и система сохраняет ее в стеке (называется *BackStack*).

Когда одна активность запускает другую, в жизненных циклах обеих из них происходит переход из одного состояния в другое. Первая активность приостанавливается и завершается (однако она не будет остановлена, если по-прежнему видима на фоне), а вторая активность создается. В случае если эти операции обмениваются данными, сохраненными на диске или в другом месте, первая *Activity* не останавливается полностью до тех пор, пока не будет создана вторая *Activity*. Наоборот, процесс запуска второй накладывается на процесс остановки первой. Порядок обратных вызовов жизненного цикла четко определен, в частности, когда в одном и том же процессе находятся две *Activity* и одна из них запускает другую.

Добавим к проекту новую *Activity* (рис. 4.1, рис. 4.2).

Если открыть файл манифеста *AndroidManifest.xml*, то можем найти там определение второй активности:

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".SecondActivity"></activity>
```

```
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

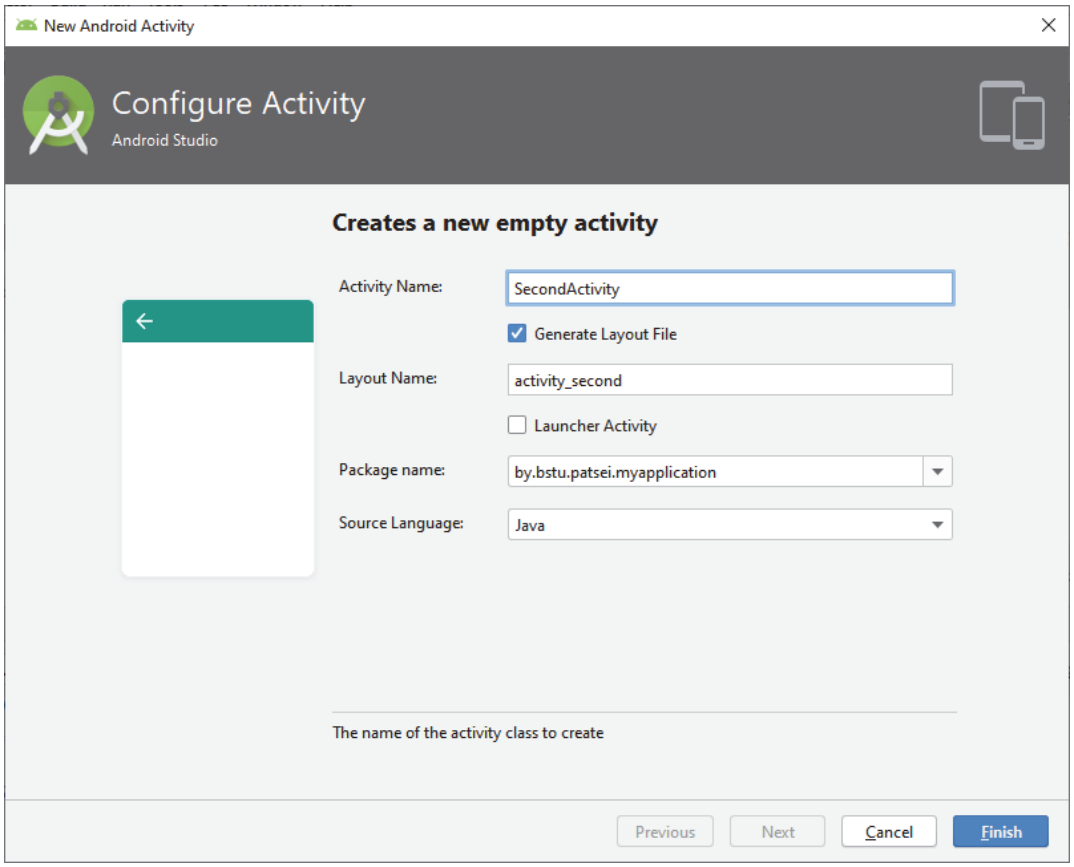


Рис. 4.1. Окно создания новой *Activity*

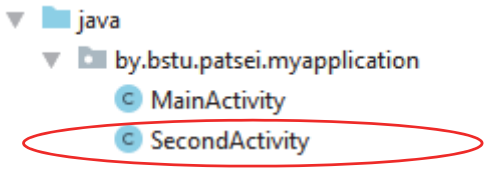


Рис. 4.2. Проект с двумя *Activity*

В *<action>* значение *android.intent.action.MAIN* представляет главную точку входа в приложение. Для *SecondActivity* просто указано, что она используется в проекте.

Если приложение будет самодостаточным и запрещено другим приложениям активировать его, то других фильтров намерений (intent-filter) создавать не нужно. Только в одной активности должно иметься «основное» действие, и ее следует поместить в категорию средств запуска, как в данном примере.

4.2. Понятие Intent

Для взаимодействия между различными объектами активности ключевым классом является *android.content.Intent*. Он представляет собой задачу, которую надо выполнить приложению.

Для запуска другой активности достаточно вызвать метод *startActivity*(), передав в него объект *Intent*, который ее описывает. В *Intent* либо указывается точная активность для запуска, либо описывается тип *Activity*, которую надо выполнить (после чего ОС выбирает подходящую активность, которая может находиться в другом приложении). *Intent* может содержать небольшой объем данных, которые будут использоваться запущенной *Activity*.

При работе с собственным приложением требуется лишь запустить нужную *Activity*. Для этого необходимо создать *Intent*, который явно определяет *Activity* с помощью имени класса. Например:

```
Intent intent = new Intent(this, Target.class);
startActivity(intent);
```

При создании *Intent* используется конструктор с двумя параметрами. Первый параметр – это *Context*. Здесь *Activity* является подклассом *Context* (объект, который предоставляет доступ к базовым функциям приложения, таким как доступ к ресурсам и файловой системе, вызов *Activity* и т. д.). Второй параметр – имя класса.

4.2.1. Передача и получение значений из Activity

При переходе от одной активности к другой можно передавать различную информацию с помощью метода *putExtra*():

```
intent.putExtra("имя", значение);
```

Фактически это словарь, который состоит из пар «ключ – значение». Можно передавать не только строковые значения, но и другие, например числовые, логические. Вызывая метод *getIntent*(), можно получить

объект *Intent*, а с помощью его метода *getExtras*() – те данные, которые ранее были переданы с объектом *Intent*. Для получения строковых данных используется метод *getStringExtra*(), для получения данных типа *float* – *getFloatExtra*(), а в качестве параметра используется ключ:

```
Intent intent = getIntent();
...
String string = intent.getStringExtra("имя");
int intNum = intent.getIntExtra("имя", 0);
```

4.2.2. Запуск Activity для получения результата

После запуска активности может потребоваться получить результат. Для этого вызывается метод *startActivityForResult*() вместо *startActivity*(). Чтобы получить результат после выполнения последующей активности, необходимо реализовать метод обратного вызова *onActivityResult*(). По завершении она возвращает результат в объекте *Intent* в вызванный метод *onActivityResult*().

4.2.3. Использование Intent

Intent представляет собой объект обмена сообщениями, с помощью которого можно запросить выполнение действия у компонента другого приложения. *Intent* используется в трех ситуациях:

1) для запуска *Activity*, как было показано выше;

2) для запуска *Service* (службы). Объект *Intent* описывает службу, которую требуется запустить, а также содержит все остальные необходимые данные;

3) для рассылки широковещательных сообщений. Для выдачи широковещательных сообщений другим приложениям необходимо передать объект *Intent* методу *sendBroadcast*(), *sendOrderedBroadcast*() или *sendStickyBroadcast*().

4.3. Типы объектов Intent

Есть два типа объектов Intent: явные и неявные.

Явный объект *Intent* указывает компонент, который требуется запустить, по имени (полное имя класса). Явные объекты *Intent* обычно используются для запуска компонента из своего приложения, поскольку известно имя класса *Activity* или службы, которую необходимо запустить.

Неявный объект *Intent* не содержит имени конкретного компонента. Вместо этого он в целом объявляет действие, которое требуется выполнить, и компонент из другого приложения обработает этот запрос.

Например, если требуется показать пользователю место на карте, то с помощью неявного объекта *Intent* можно запросить, чтобы это сделало другое приложение, в котором такая возможность предусмотрена.

Когда создан *неявный* объект *Intent*, система Android находит подходящий компонент путем сравнения содержимого объекта *Intent* с фильтрами *Intent*, объявленными в файлах манифеста других приложений. Если объект *Intent* совпадает с фильтром *Intent*, система запускает этот компонент и передает ему объект *Intent*. Если подходящими оказываются несколько фильтров *Intent*, система выводит диалоговое окно, где пользователь может выбрать приложение для выполнения данного действия.

Фильтр *Intent* представляет собой выражение в файле манифеста приложения, указывающее типы объектов *Intent*, которые мог бы принимать компонент. Например, при объявлении фильтра *Intent* для активности другим приложениям дается возможность напрямую запускать *Activity* с помощью некоторого объекта *Intent*. Точно так же, если не объявить какие-либо фильтры *Intent* для активности, ее можно будет запустить только с помощью явного объекта *Intent*.

4.3.1. Задание неявного объекта Intent

В приложении может потребоваться выполнить некоторое действие, например отправить письмо, текстовое сообщение или обновить статус, используя данные из текущей *Activity*. В этом случае можно воспользоваться *Activity* из других приложений, имеющихся на устройстве.

Например, если пользователю требуется предоставить возможность отправить электронное письмо, можно создать следующий *Intent*:

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType("text/plain");

// Проверка возможности выполнения
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(sendIntent);
}
```

Дополнительный компонент *EXTRA_*, добавленный в *Intent*, представляет собой текст письма. Когда почтовая программа реагирует на *Intent*, она считывает дополнительно добавленную строку и помещает ее в поле текста письма. При этом запускается активность почтовой программы, а после того, как пользователь завершит требуемые действия, возобновляется текущая активность.

4.3.2. Определение объекта Intent

Рассмотрим сведения, которые содержит *Intent*.

Имя компонента, который требуется запустить. Эта информация является необязательной, но именно она и делает объект *Intent* ***явным***. При отсутствии имени компонента объект *Intent* является ***неявным***. Имя можно задать через объект *ComponentName*(), *setComponent*(), *setClass*(), *setClassName*() или конструктор *Intent.*

Действие – это строка, определяющая стандартное действие, которое требуется выполнить (*view*, *pick*, ...). Действие в значительной степени определяет, каким образом будет структурирована остальная часть объекта *Intent*. Для использования *Intent* в пределах своего приложения можно указать собственные действия. Обычно следует использовать константы действий, определенные классом *Intent* или другими классами платформы (например, *ACTION_VIEW*, *ACTION_SEND* и др.). Задается методом *setAction*() или конструктором *Intent*.

Данные – это объект URI, ссылающийся на данные, с которыми будет выполняться действие и (или) тип MIME этих данных. Тип данных определяется действием объекта *Intent*. Например, если действием является *ACTION_EDIT*, в данных должен содержаться URI документа, который требуется отредактировать. Чтобы задать только URI данных, вызывается *setData*(). Чтобы задать только тип MIME – *setType*(). При необходимости определить оба этих параметра – *setDataAndType*().

Категория – строка, содержащая прочие сведения о том, каким компонентом должна выполняться обработка этого объекта *Intent*. В объект *Intent* можно поместить любое количество описаний категорий, но большинству объектов *Intent* категория не требуется. Например, стандартные категории: *CATEGORY_BROWSABLE* (целевая активность позволяет запускать себя веб-браузером для отображения данных, указанных по ссылке); *CATEGORY_LAUNCHER* (активность является начальной для задачи). Указать категорию можно с помощью *addCategory*().

Дополнительные данные – это пары «ключ – значение», содержащие прочую информацию. Добавлять дополнительные данные можно с помощью методов *putExtra*(), каждый из которых принимает два параметра: имя и значение ключа. Можно создать объект *Bundle* со всеми дополнительными данными, затем вставить объект *Bundle* в объект *Intent* с помощью метода *putExtras*().

Класс *Intent* указывает много констант *EXTRA_** для стандартных типов данных. Если требуется объявить собственные дополнительные ключи, то в качестве префикса указывается имя пакета приложения.

Флаги, определенные в классе *Intent*, действуют как метаданные. Флаги должны указывать ОС, каким образом следует запускать активность (например, к какой задаче должна принадлежать активность и как с ней обращаться после запуска).

Возможна ситуация, когда на устройстве пользователя не будет никакого приложения, которое может откликнуться на неявный объект *Intent*. В этом случае вызов закончится неудачей, а работа приложения аварийно завершится. Чтобы проверить, будет ли получен объект *Intent*, необходимо вызвать метод *resolveActivity*() для объекта *Intent*. Если результатом будет значение, отличное от *null*, значит, имеется хотя бы одно приложение, которое способно откликнуться на объект *Intent*, поэтому можно вызывать *startActivity*().

Рассмотрим пример определения неявного объекта *Intent*:

```
Button sbut = (Button) findViewById(R.id.bSend);
sbut.setOnClickListener(new View.OnClickListener() {

  @Override
   public void onClick(View view) {
        Intent sendIntent = new Intent();
        sendIntent.setAction(Intent.ACTION_SEND);
        sendIntent.putExtra(Intent.EXTRA_TEXT, "Send something here");
        sendIntent.setType("text/plain");

        Intent chooser = Intent.createChooser(sendIntent,
                              "Choose the activity");
        if (sendIntent.resolveActivity(getPackageManager()) != null) {
                    startActivity(chooser);
                }
            }
});
```

4.4. Настройка фильтров для объекта Intent

Чтобы указать, какие неявные объекты *Intent* может принимат приложение, следует объявить один или несколько фильтров *Intent* для каждого компонента приложения с помощью элемента *intent-filter* в файле манифеста. Система передаст неявный объект *Intent* приложению, только если он может пройти через один из фильтров.

Явный объект *Intent* всегда доставляется целевому компоненту, без учета любых фильтров *Intent*, объявленных компонентом.

Компонент приложения должен объявлять отдельные фильтры для каждой уникальной работы, которую он может выполнить. Каждый фильтр *Intent* определяется элементом *intent-filter* в файле манифеста

приложения, указанном в объявлении соответствующего компонента приложения. Внутри элемента *intent-filter* можно указать тип объектов *Intent* с помощью одного или нескольких элементов: *action* – объявляет принимаемое действие, заданное в объекте *Intent*, в атрибуте *name*; *data* – объявляет тип принимаемых данных; *category* – объявляет принимаемую категорию, заданную в объекте *Intent*, в атрибуте *name*:

```
<intent-filter>
        <action>
        </action>

        <data>
        </data>

        <category>
        </category>
</intent-filter>
```

Можно создавать фильтры, в которых будет несколько экземпляров *action*, *data*, *category*. В этом случае просто нужно убедиться в том, что компонент может справиться с любыми сочетаниями этих элементов фильтра.

Чтобы объект *Intent* был доставлен компоненту, он должен пройти все три теста. Если он не будет соответствовать хотя бы одному из них, ОС не доставит этот объект *Intent* компоненту. Поскольку у компонента может быть несколько фильтров, объект *Intent*, который не проходит ни через один из них, может пройти через другой фильтр.

4.5. Разрешение объектов Intent

Когда система получает неявный объект *Intent* для запуска активности, она выполняет поиск путем сравнения объекта *Intent* с фильтрами по трем критериям: действие объекта *Intent*; данные объекта (структура URI и тип данных); категория объекта.

4.5.1. Тестирование действия

Для указания принимаемых действий объекта *Intent* фильтр может объявлять любое (в том числе нулевое) число элементов *action*:

```
<intent-filter>
    <action android:name="android.intent.action.EDIT" />
    <action android:name="android.intent.action.VIEW" />
    ...
</intent-filter>
```

Чтобы пройти через этот фильтр, действие, указанное в объекте *Intent*, должно соответствовать одному или нескольким действиям, перечисленным в фильтре. Если в фильтре не перечислены действия, объекту *Intent* будет нечему соответствовать, поэтому все объекты *Intent* не пройдут этот тест. Если в объекте *Intent* не указано действие, он пройдет тест.

Действия задаются константами действий:

— *ACTION_ANSWER* – открывает активность, которая связана с входящими звонками;

— *ACTION_CALL* – инициализирует обращение по телефону;

— *ACTION_DELETE* – запускает активность, с помощью которой можно удалить данные, указанные в пути URI внутри *Intent*;

— *ACTION_EDIT* – отображает данные для редактирования пользователем;

— *ACTION_INSERT* – открывает активность для вставки в *Cursor* нового элемента, указанного с помощью пути URI;

— *ACTION_HEADSET_PLUG* – подключение наушников;

— *ACTION_MAIN* – запускается как начальная активность задания;

— *ACTION_PICK* – загружает дочернюю *Activity*, позволяющую выбрать элемент из источника данных, указанный с помощью URI;

— *ACTION_SEARCH* – запускает активность для выполнения поиска;

— *ACTION_SEND* – загружает экран для отправки данных, указанных в намерении;

— *ACTION_SENDTO* – открывает активность для отправки сообщений контакту, указанному в пути URI, который передается через *Intent*;

— *ACTION_SYNC* – синхронизирует данные сервера с данными мобильного устройства;

— *ACTION_VIEW* – наиболее распространенное общее действие.

4.5.2. Тестирование категории

Для указания принимаемых категорий фильтр *Intent* может объявлять любое (в том числе нулевое) число элементов *category*:

```
<intent-filter>
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    ...
</intent-filter>
```

Чтобы объект *Intent* прошел тестирование категории, все категории, приведенные в объекте *Intent*, должны соответствовать категории

из фильтра. Обратное не требуется – фильтр *Intent* может объявлять и другие категории, которых нет в объекте *Intent*, при этом он все равно пройдет тест. Поэтому объект *Intent* без категорий всегда пройдет этот тест, независимо от того, какие из них объявлены в фильтре.

Система Android автоматически применяет категорию *CATEGORY_DEFAULT* ко всем неявным объектам *Intent*, которые передаются в *startActivity*() и *startActivityForResult*(). Поэтому, если нужно, чтобы *Activity* принимала неявные объекты *Intent*, в ее фильтрах должна быть указана категория для *android.intent.category.DEFAULT*.

Можно задать собственные категории или же брать стандартные значения, предоставляемые системой:

– *ALTERNATIVE* – действие должно быть доступно в качестве альтернативного тому, которое выполняется по умолчанию для элемента этого типа данных. Например, если действие по умолчанию для контакта – просмотр, то в качестве альтернативы его также можно редактировать;

– *SELECTED_ALTERNATIVE* – то же самое, что и *ALTERNATIVE*, но вместо одиночного действия применяется в тех случаях, когда нужен список различных возможностей;

– *BROWSABLE* – действие доступно из браузера;

– *DEFAULT* – делает компонент обработчиком по умолчанию для действия, выполняемого с указанным типом данных внутри фильтра;

– *GADGET* – активность может запускаться внутри другой активности;

– *LAUNCHER* – помещает *Activity* в окно для запуска приложений.

4.5.3. Тестирование данных

Для указания принимаемых данных объекта *Intent* фильтр может объявлять любое (в том числе нулевое) число элементов *data*. Например:

```
<intent-filter>
    <data android:mimeType="video/mpeg" android:scheme="http" ... />
    <data android:mimeType="audio/mpeg" android:scheme="http" ... />
    ...
</intent-filter>
```

Каждый элемент *data* может конкретизировать структуру URI и тип данных. Имеются отдельные атрибуты – *scheme*, *host*, *port* и *path* – для каждой составной части URI: *scheme://host:port/path*.

Каждый из этих атрибутов является необязательным, но есть линейные зависимости: если схема не указана, узел игнорируется; если узел не указан, порт игнорируется; если не указана ни схема, ни узел, путь игнорируется.

4.6. Принцип работы фильтров

В целом весь алгоритм работает следующим образом. Android собирает список всех доступных фильтров из установленных пакетов.

Фильтры, которые не соответствуют действию или категории *Intent*, удаляются из списка. Совпадение происходит только в том случае, если фильтр содержит указанное действие (или если действие для него не задано).

Для категорий процесс соответствия более строгий. Фильтр должен включать в себя все категории, заданные в полученном *Intent*. Фильтр, для которого категории не указаны, может соответствовать только таким же объектом *Intent* (нет категорий).

Каждая часть пути URI из *Intent* сравнивается с тегом *data*. Если в фильтре указаны схема (протокол), сервер/принадлежность, путь или тип MIME, все эти значения проверяются на соответствие пути URI из *Intent*. При любом несовпадении фильтр будет удален из списка. Если в фильтре не указано ни одного параметра *data*, его действие будет распространяться на любые данные.

MIME – тип данных, который должен совпасть. При сравнении типов данных можно использовать маски, чтобы охватывать все подтипы. Если в фильтре указан тип данных, он должен совпасть с тем, который значится в *Intent*, при отсутствии тега *data* подойдет любой тип.

Схема – это протокольная часть пути URI, например http:, mailto: или tel:.

Имя сервера (или принадлежность данных) – часть URI между схемой и самим путем.

5. БАЗОВЫЕ ЭЛЕМЕНТЫ НАВИГАЦИИ

5.1. Меню

Начиная с версии Android 3.0 (уровень API 11) меню переместилось в *ActionBar*. Существуют следующие виды меню:

1) ***меню параметров и строка действий*** (*ActionBar*) – пункты меню параметров размещаются в строке действий в виде сочетания отображаемых на экране вариантов действий и раскрывающегося списка дополнительных вариантов выбора (рис. 5.1);

Рис. 5.1. Меню параметров и строка действий

2) ***контекстное меню и режим контекстных действий*** – это плавающее меню, которое открывается, когда пользователь длительно нажимает на элемент. В режиме контекстных действий в строке, расположенной вверху экрана, отображаются пункты действий, затрагивающие выбранный контент, причем пользователь может выбрать сразу несколько элементов;

3) ***всплывающее меню*** – отображается вертикальный список пунктов, который привязан к представлению, вызвавшему меню.

5.1.1. Определение меню в файле XML

Для определения пунктов меню используется стандартный формат XML в ресурсе меню. После этого ресурс меню можно будет загружать как объект *Menu* в активности или фрагменты.

Использовать меню в ресурсах рекомендуется по нескольким причинам:

1) в XML проще визуализировать структуру;

2) позволяет отделить контент меню от кода, определяющего работу приложения;

3) позволяет создавать альтернативные варианты меню для разных версий платформы, размеров экрана и других конфигураций.

Чтобы определить меню, создается файл в папке *res/menu/имя.xml* проекта и определяется меню со следующими элементами (результат выполнения представлен на рис. 5.2):

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

    <item android:id="@+id/newA"
        android:icon="@drawable/metro_android2_black"
        android:title="@string/newel"/>

    <item android:id="@+id/nextB"
        android:icon="@drawable/beret"
        android:title="@string/nextel" />

</menu>
```

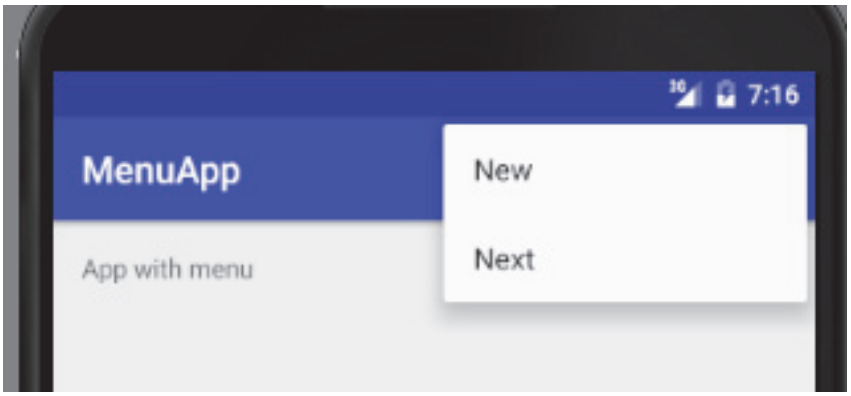


Рис. 5.2. Пример приложения с меню

В файле описания меню:

— *<menu>* – определяет класс *Menu*, который является контейнером для пунктов. Элемент *<menu>* должен быть корневым узлом файла, в котором может находиться один или несколько элементов *<item>* и *<group>*;

— *<item>* – создает класс *MenuItem*, который представляет один пункт меню. Этот элемент может содержать вложенный элемент *<menu>* для создания вложенных меню:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:id="@+id/file"
        android:title="@string/file" >
        <!-- вложенное меню -->
        <menu>
```

```
        <item android:id="@+id/create_new"
            android:title="@string/create_new" />

        <item android:id="@+id/open"
            android:title="@string/open" />

    </menu>
 </item>

</menu>
```

— *<group>* – необязательный, невидимый контейнер для элементов *<item>*. Он позволяет разделять пункты меню на категории и назначать им одинаковые свойства, такие как активное состояние и видимость. Содержит *<item>* и должен быть дочерним от *<menu>*.

Элемент *<item>* поддерживает несколько атрибутов, с помощью которых можно определить внешний вид и поведение. Пункты меню чаще всего имеют следующие атрибуты:

— *android*:*id* – идентификатор ресурса;

— *android*:*icon* – ссылка на графический элемент, который будет использоваться в качестве значка пункта меню;

— *android*:*title* – ссылка на строку, которая будет использоваться в качестве названия пункта меню;

— *android*:*orderInCategory* – порядок следования элемента в меню;

— *android*:*showAsAction* – указывает, когда и как этот пункт должен отображаться в строке действий. Принимает следующие значения:

```
["ifRoom" | "never" | "withText" | "always" | "collapseActionView"]
```

— *ifRoom* – добавить, если есть место для него, если нет, то для всех элементов с надписью "ifRoom", элементы с наименьшими значениями *OrderInCategory* отображаются как действия, а остальные элементы помещаются в меню переполнения;

— *withText* – включает текст заголовка;

— *never* – никогда не следует помещать этот элемент в *ActionBar*;

— *always* – всегда размещать этот элемент в *ActionBar*.

```
<item android:id="@+id/newA"
android:icon="@drawable/metro_android2_black"
        android:title="@string/newel"/>
        <!--android:showAsAction="ifRoom"/>-->
...
```

Место, где отображаются на экране пункты меню параметров, определяется версией платформы, для которой разработано приложение.

Если приложение написано для версии Android 2.3.x (уровень API 10) или более ранней, содержимое меню параметров отображается внизу экрана, когда пользователь нажимает кнопку *Меню*. Если приложение предназначено для версии Android 3.0 (уровень API 11) и более поздних, пункты меню параметров будут отображаться в строке действий. Чтобы обеспечить быстрый доступ к важным действиям, можно принудительно разместить несколько пунктов меню в строке действий, добавив *android*:*showAsAction*="*ifRoom*" к соответствующим элементам (рис. 5.3).

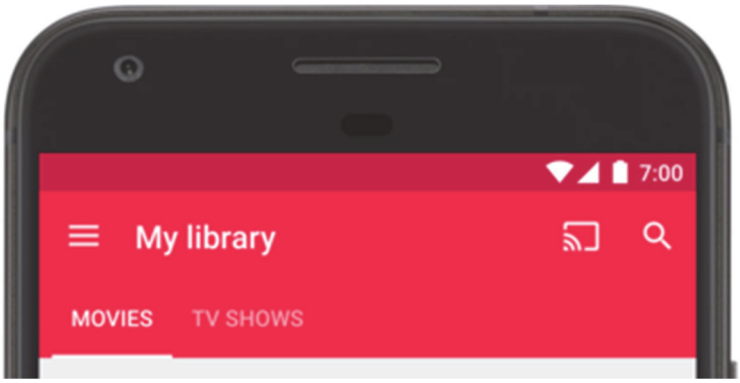


Рис. 5.3. Пример *ActionBar*

В современных устройствах меню является частью *ActionBar*.

5.1.2. Вывод меню на экран

Мы определили меню, но само определение элементов в файле еще не создает меню. Это лишь декларативное описание. Чтобы вывести его на экран, надо вызвать его в классе *Activity*. Для этого следует переопределить метод *onCreateOptionsMenu*:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {

   MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu_ops, menu);
    return true;
}
```

Программа должна сконвертировать созданный ресурс меню в программный объект. Для этой цели существует специальный метод *MenuInflater.inflate*(). Он предназначен для вывода меню при нажатии кнопки на устройстве. В качестве первого параметра метод принимает ресурс, представляющий декларативное описание меню в XML, и наполняет им объект *Menu*, переданный в качестве второго параметра. Метод должен возвращать значение *true*, чтобы меню было видимым на экране.

Метод *getMenuInflater*() возвращает экземпляр класса *MenuInflater*, который используется для чтения данных меню из XML.

Одновременно можно вывести на экран шесть пунктов меню. Если пунктов больше, то будет выведено пять плюс шестой пункт *More*, который позволит увидеть остальные пункты при нажатии.

5.1.3. Обработка нажатий

Когда пользователь выбирает пункт меню (в том числе пункты из *строки действий*), система вызывает метод *onOptionsItemSelected*() активности. Этот метод передает в параметрах класс *MenuItem*. Например, результат работы фрагмента листинга представлен на рис. 5.4.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {

    // TODO
    Toast.makeText(this, item.getTitle(), Toast.LENGTH_LONG).show();
    return super.onOptionsItemSelected(item);
}
```

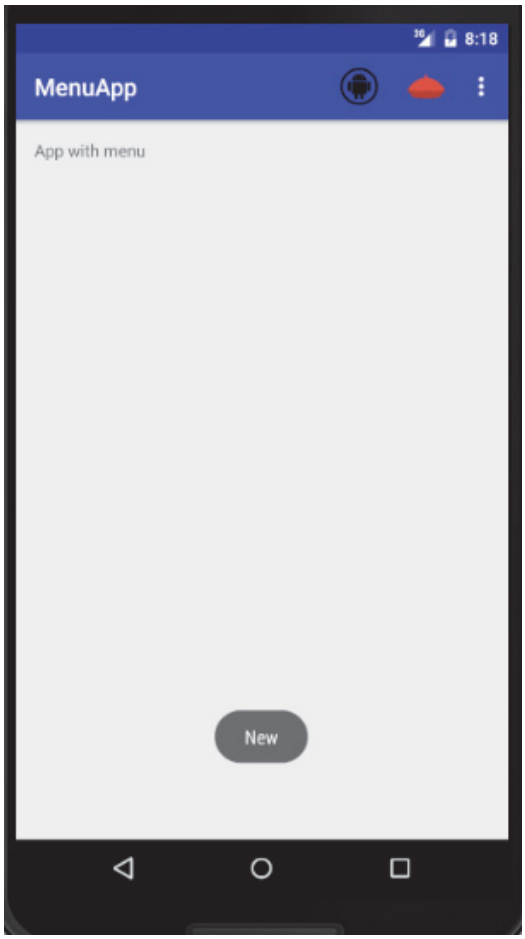


Рис. 5.4. Обработка пунктов меню

Когда пункт меню успешно обработан, возвращается *true*. Если пункт меню не обрабатывается, следует вызвать реализацию суперкласса *onOptionsItemSelected*() (реализация по умолчанию возвращает значение *false*).

Идентифицировать пункт можно путем вызова метода *getItemId*(), который возвращает уникальный идентификатор пункта меню (определен атрибутом *android:id* из ресурса меню). Этот идентификатор можно сопоставить с известными пунктами, чтобы выполнить соответствующее действие:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {

        // TODO
        switch (item.getItemId()) {
            case R.id.newA:
                newA();
                return true;
            case R.id.nextB:
                next();
                return true;
            case R.id.saveC:
                saveC();
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
}
```

Если в активности имеются фрагменты, система сначала вызовет метод *onOptionsItemSelected*(), а затем будет вызывать этот метод для каждого фрагмента (в том порядке, в котором они были добавлены), пока он не возвратит значение *true* или не закончатся фрагменты. Меню можно обработать следующим образом:

```
MenuItem mItem=menu.findItem(R.id.newA);
mItem.setOnMenuItemClickListener(

        new MenuItem.OnMenuItemClickListener() {
            @Override
            public boolean onMenuItemClick(MenuItem item) {
               // ....
               return false;
            }
        });
```

5.1.4. Программное создание меню и изменение пунктов меню во время выполнения приложения

Если требуется изменять меню в зависимости от событий, которые возникают в течение жизненного цикла активности, сделать это можно в методе *onPrepareOptionsMenu*(). Этот метод вызывается каждый раз,

когда меню отображается или перерисовывается. Он передает объект *Menu* в том виде, в котором он в данный момент существует. Его можно изменить путем, например, добавления, удаления или отключения пунктов меню:

```
@Override
public boolean onPrepareOptionsMenu (Menu menu){

      menu.add("Delete");
      menu.add("Move");
      //  menu.removeItem(R.id.nextB);

return super.onPrepareOptionsMenu(menu);
}
```

У метода *add*() могут быть четыре параметра: *идентификатор группы* – позволяет связывать пункт меню с группой других пунктов этого меню; *идентификатор пункта* – для обработчика события выбора пункта меню; *порядок расположения пункта в меню* – позволяет определять позицию в меню, по умолчанию (*Menu.NONE* или 0) пункты идут в том порядке, как задано в коде; *заголовок* – текст, который выводится в пункте меню (результат выполнения представлен на рис. 5.5).

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {

    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu_ops, menu);
    MenuItem menuItem = menu.add(Menu.NONE,111, Menu.NONE, "Move");
    menuItem.setShowAsAction(MenuItem.SHOW_AS_ACTION_ALWAYS);
    return true;
}
```

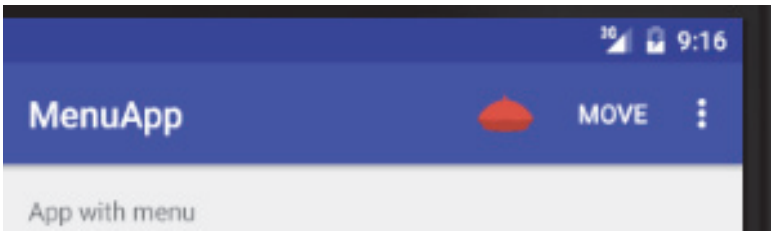


Рис. 5.5. Пример добавления пунктов меню в коде

5.1.5. Создание контекстного меню

Кроме стандартного меню в Android используется также *контекстное меню*, вызываемое при нажатии на объект в течение 2–3 с (событие long-press). В отличие от обычного меню в контекстном не поддерживаются значки и быстрые клавиши. Второе важное отличие –

контекстное меню применимо к *View*, а меню к *Activity*. Поэтому в приложении может быть одно меню и несколько контекстных меню, например у каждого элемента *TextView*.

Существует два способа предоставления возможности контекстных действий:

1) ***в плавающем контекстном меню.*** При этом меню отображается в виде плавающего списка пунктов меню (наподобие диалогового окна). Пользователи могут каждый раз выполнять контекстное действие только с одним элементом (представлено на рис. 5.6);

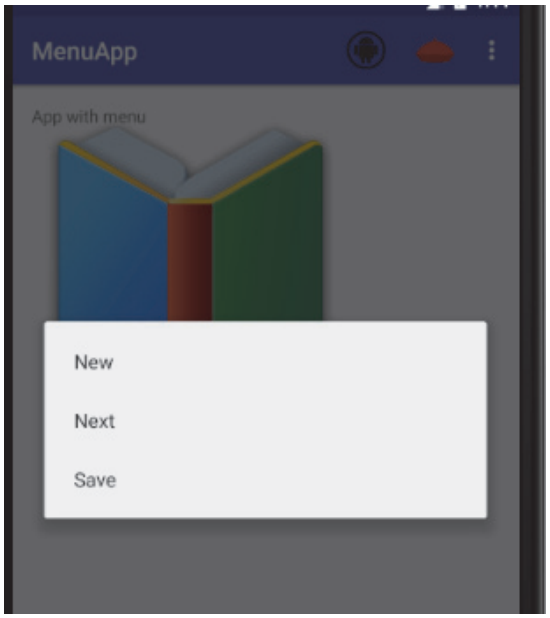


Рис. 5.6. Плавающее контекстное меню

2) ***в режиме контекстных действий*** (представлен на рис. 5.7). Этот режим является системной реализацией *ActionMode*, которая отображает ***строку контекстных действий*** вверху экрана с пунктами действий, которые затрагивают выбранные элементы. Когда этот режим активен, пользователи могут одновременно выполнять действие с несколькими элементами (если это допускается приложением). Если данный режим предусмотрен, именно ***его рекомендуется использовать*** для отображения контекстных действий.

Рассмотрим алгоритм ***создания плавающего контекстного меню***. Вначале зарегистрируем класс *View*, с которым следует связать контекстное меню, вызвав метод *registerForContextMenu*() и передав ему *View*:

```
setContentView(R.layout.activity_menu);
ImageView img=(ImageView) findViewById(R.id.image_book);
registerForContextMenu(img);
```

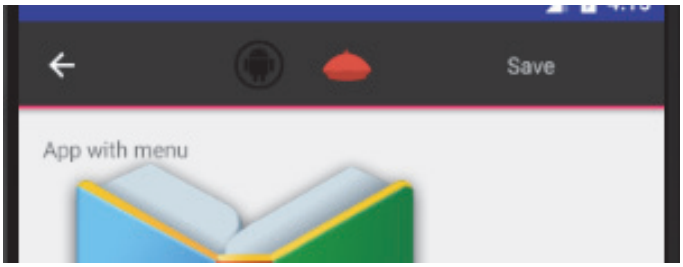


Рис. 5.7. Плавающее меню
в режиме контекстных действий

Если активность использует *ListView* или *GridView*, требуется, чтобы каждый элемент предоставлял одинаковое контекстное меню. Следует зарегистрировать все элементы для контекстного меню. Затем реализовать метод *onCreateContextMenu*() в активности или фрагменте:

```
@Override
public void onCreateContextMenu( ContextMenu menu,View v,
                                 ContextMenu.ContextMenuInfo menuInfo) {

    super.onCreateContextMenu(menu, v, menuInfo);
    getMenuInflater().inflate(
            R.menu.menu_ops,menu);
}
```

Когда зарегистрированное представление примет событие длительного нажатия, система вызовет метод *onCreateContextMenu*(). Именно здесь определяются пункты меню. Делается это обычно путем загрузки ресурса меню. Реализуется метод *onContextItemSelected*(). Когда пользователь выбирает пункт меню, система вызывает этот метод, с тем чтобы можно было выполнить соответствующее действие:

```
@Override
public boolean onContextItemSelected(

    MenuItem item) {
    switch (item.getItemId()) {
        case R.id.newA:
            newA();
            return true;
        case R.id.nextB:
            next();
            return true;
        case R.id.saveC:
            saveC();
            return true;
```

```
        default:
            return super.onContextItemSelected(item);
    }
}
```

Рассмотрим вариант реализации ***режима контекстных действий***. Он представляет собой системную реализацию класса *ActionMode*. Когда пользователь использует этот режим, выбирая элемент, вверху экрана открывается ***строка контекстных действий***, содержащая действия, которые пользователь может выполнить с выбранными в данный момент элементами. Режим контекстных действий отключается, а строка контекстных действий исчезает, когда пользователь снимет выделение со всех элементов, нажмет кнопку *Назад* или выберет действие *Готово*, расположенное с левой стороны строки.

Строка контекстных действий не обязательно бывает связана со строкой действий. Они работают независимо друг от друга, даже несмотря на то, что визуально строка контекстных действий занимает положение строки действий.

Для включения режима контекстных действий для отдельных представлений необходимо сделать следующее:

1) захватить элемент и добавить обработчик на длительное нажатие:

```
EditText etx=(EditText)findViewById(R.id.edit_text);

etx.setOnLongClickListener(
        new View.OnLongClickListener() {
            @Override
            public boolean onLongClick(View v) {
                return false;
            }
        }
);
```

2) реализовать интерфейс *ActionMode.Callback*. В его методах обратного вызова можно указывать действия для строки контекстных действий, реагировать на нажатия пунктов и обрабатывать другие события жизненного цикла для режима действий:

```
private ActionMode.Callback
        mActionModeCallback=new ActionMode.Callback(){

    @Override
    public boolean onCreateActionMode(ActionMode actionMode, Menu
menu) {
        return false;
    }
```

```
    @Override
    public boolean onPrepareActionMode(ActionMode actionMode, Menu
menu) {
        return false;
    }
    @Override
    public boolean onActionItemClicked(ActionMode actionMode, MenuItem
menuItem) {
        return false;
    }

    @Override
    public void onDestroyActionMode(ActionMode actionMode) {
      actionMode = null;
    }
};
```

Например, может быть такая реализация (в случае если меню определено в ресурсах):

```
@Override
public boolean onCreateActionMode(ActionMode actionMode, Menu menu) {

    actionMode.getMenuInflater().inflate(
            R.menu.menu_ops, menu);
    return true;
}
```

3) вызывать *startActionMode*(), когда требуется показать строку (например, когда пользователь выполняет длительное нажатие представления):

```
EditText etx=(EditText)findViewById(R.id.edit_text);
etx.setOnLongClickListener(
        new View.OnLongClickListener() {

            @Override
            public boolean onLongClick(View v) {
               startActionMode(mActionModeCallback);
                v.setSelected(true);
                return true;
            }
        });
```

При вызове метода *startActionMode*() система возвращает созданный класс *ActionMode*. Сохранив его в составной переменной, можно вносить изменения в строку контекстных действий в ответ на другие события (рис. 5.8).

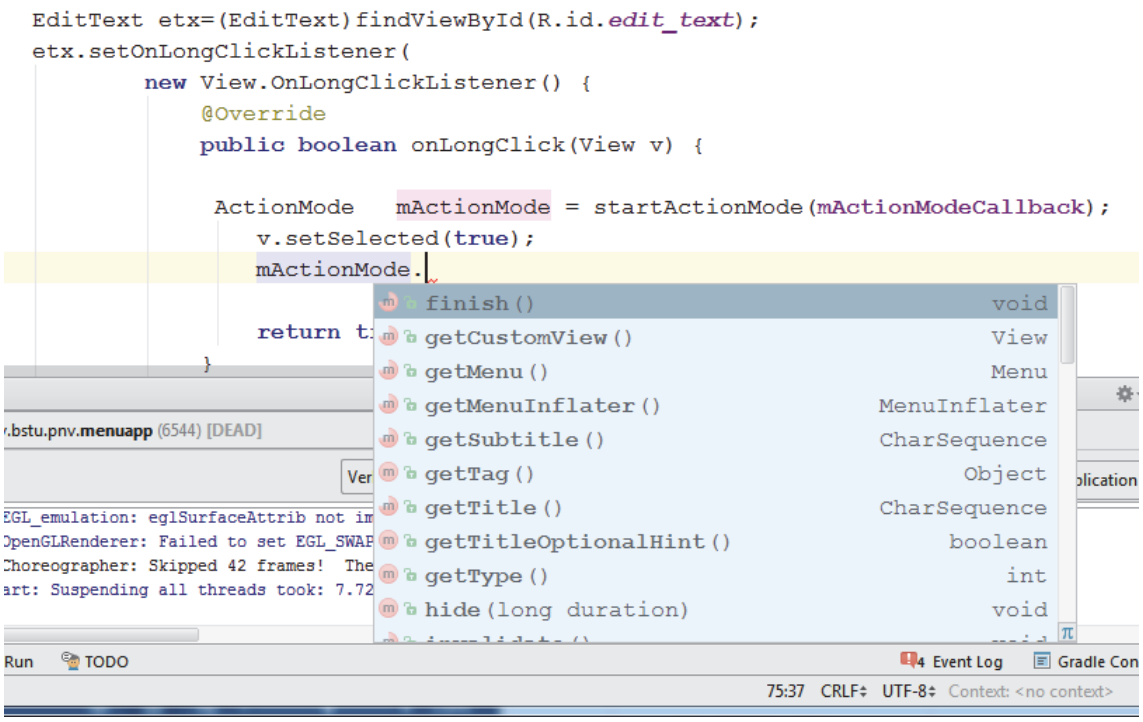


Рис. 5.8. Управление строкой контекстных действий

5.1.6. Создание всплывающего меню

Всплывающее, или *PopupMenu*, является модальным меню, привязанным к *View*. Оно отображается ниже представления, к которому привязано, если там есть место, либо поверх него (рис. 5.9).

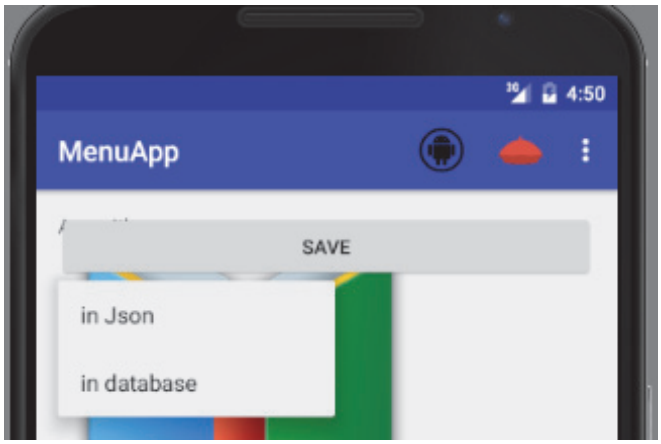


Рис. 5.9. Всплывающее меню

Для определения меню используется XML:

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

<menu xmlns:android="http://schemas.android.com/apk/res/android">
```

```
    <item android:id="@+id/sJ"
          android:title="@string/in_json"/>

    <item android:id="@+id/sDB"
          android:title="@string/in_database"/>

</menu>
```

В этом случае:

1) создается экземпляр класса *PopupMenu* с помощью конструктора, принимающего текущие приложения *Context* и *View*;

2) с помощью *MenuInflater* загружается ресурс меню в объект *Menu*, возвращенный методом *PopupMenu.getMenu*(). На API 14-го уровня и выше вместо этого можно использовать *PopupMenu.inflate*();

3) вызывается метод *PopupMenu.show*():

```
Button button = (Button) findViewById(R.id.btn_popup);
button.setOnClickListener(
        new View.OnClickListener() {

            @Override
            public void onClick(View v) {
                PopupMenu popup = new PopupMenu(v.getContext(),v);
                popup.inflate(R.menu.menu_popup_btn);
                popup.show();
            }
        });
```

5.1.7. Создание приложения Блокнот

Для создания приложения – аналога блокнота, позволяющего записывать и читать данные только из одного фиксированного файла, на экране активности можно разместить компонент *EditText* на весь экран. Затем заменить файл *activity_main.xml* следующим содержимым:

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

    <EditText
        android:id="@+id/editText"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="top|left"
        android:inputType="textMultiLine" />

</LinearLayout>
```

Для создания меню, содержащего два пункта *Открыть* и *Сохранить*, понадобятся иконки. Для этого есть много ресурсов с иконками (в свободном доступе). Например, https://material.io/icons/. Скачаем иконки и разместим файлы в папке *drawable* (рис. 5.10):

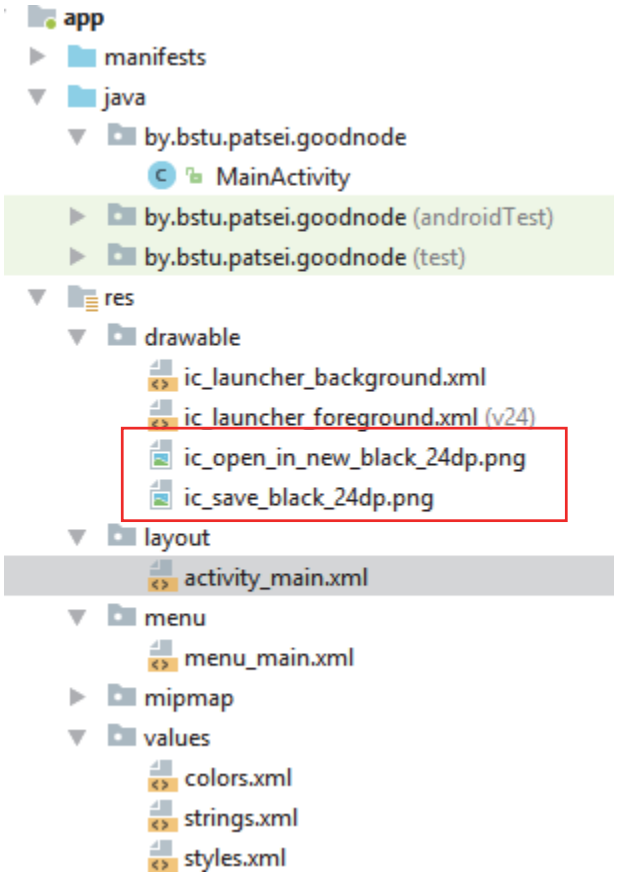


Рис. 5.10. Размещение иконок в проекте

Следует создать еще один файл с пунками меню, определенными в *res/menu/menu_main.xml*, и со следующим содержимым:

```
<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_open"
        android:icon="@drawable/ic_open_in_new_black_24dp"
        android:orderInCategory="100"
        app:showAsAction="ifRoom|withText"
        android:title="@string/action_open" />

    <item
        android:id="@+id/action_save"
```

```
        android:icon="@drawable/ic_save_black_24dp"
        android:orderInCategory="100"
        app:showAsAction="ifRoom|withText"
        android:title="@string/action_save" />

</menu>
```

Как видно, здесь идет обращение к строковым ресурсам. Поэтому необходимо изменить содержимое файла *string.xml*:

```
<resources>

        <string name="app_name">GoodNode</string>
        <string name="action_open">Открыть</string>
        <string name="action_save">Сохранить</string>
</resources>
```

Затем нужно добавить меню и для методов обработчиков написать *openFile*() и *saveFile*(), в которых реализуются операции по открытию и сохранению файла. Содержимое класса *MainActivity* будет следующим:

```
public class MainActivity extends AppCompatActivity {

    private final static String FILENAME = "sample.txt"; // Имя файла
    private EditText mEditText;

    @Override
    public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
            // Окно редактора
            mEditText = (EditText) findViewById(R.id.editText);
        }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
            getMenuInflater().inflate(R.menu.menu_main, menu);
            return true;
        }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
            // Выбор и обработка пунктов меню
            switch (item.getItemId()) {
                case R.id.action_open:
                    openFile(FILENAME);
                    return true;
                case R.id.action_save:
                    saveFile(FILENAME);
                    return true;
                default:
                    return true;
            }
        }
```

```
// Метод для открытия файла
private void openFile(String fileName) {
try {
     InputStream inputStream = openFileInput(fileName);
     if (inputStream != null) {
       InputStreamReader isr = new InputStreamReader(inputStream);
        BufferedReader reader = new BufferedReader(isr);
        String line;
        StringBuilder builder = new StringBuilder();
        while ((line = reader.readLine()) != null) {
                builder.append(line + "\n");
                }
         inputStream.close();
         mEditText.setText(builder.toString());
            }
    } catch (Throwable t) {
       Toast.makeText(getApplicationContext(),
        "Exception: " + t.toString(), Toast.LENGTH_LONG).show();
        }
}

// Метод для сохранения файла
private void saveFile(String fileName) {
try {
   OutputStream outputStream = openFileOutput(fileName, 0);
   OutputStreamWriter osw = new OutputStreamWriter(outputStream);
            osw.write(mEditText.getText().toString());
            osw.close();
    } catch (Throwable t) {
         Toast.makeText(getApplicationContext(),
         "Exception: " + t.toString(), Toast.LENGTH_LONG).show();
        }
    }
}
```

После этого надо запустить приложение. Результат запуска представлен на рис. 5.11.

5.2. Диалоговые окна

В некоторых случаях требуется отобразить диалоговое окно, где пользователю нужно сделать какой-нибудь выбор или вывести сообщение об ошибке. В Android уже есть собственные встроенные диалоговые окна, которые гибко настраиваются под задачи. Использование диалоговых окон для простых задач позволяет сократить число классов в приложениии и экономить ресурсы памяти.

Диалоговые окна в Android представляют собой ***полупрозрачные плавающие активности***, частично перекрывающие родительский экран, из которого их вызвали. Как правило, они затеняют родительскую активность позади себя с помощью фильтров размывания или затемнения.

Рис. 5.11. Результат работы приложения *Блокнот*

Можно установить заголовок с помощью метода *setTitle*() и содержимое с помощью метода *setContentView*().

Android поддерживает следующие типы диалоговых окон:

– *Dialog* – базовый класс для всех типов диалоговых окон;

– *AlertDialog* – диалоговое окно с кнопками, списком, флажками или переключателями;

– *CharacterPickerDialog* – диалоговое окно, позволяющее выбрать символ с ударением, связанный с базовым символом;

– *DatePickerDialog* – диалоговое окно выбора даты с элементом *DatePicker*;

– *TimePickerDialog* – диалоговое окно выбора времени с элементом *TimePicker*.

Поскольку *ProgressDialog*, *TimePickerDialog* и *DatePickerDialog* – расширение класса *AlertDialog*, они также могут иметь командные кнопки. Если ни один из существующих типов диалоговых окон не подходит, то можно создать собственное диалоговое окно.

5.2.1. Класс Dialog

Класс *Dialog* является базовым для всех классов диалоговых окон. Каждое диалоговое окно должно быть определено внутри активности, в которой будет использоваться. Для его отображения необходимо вызвать метод *showDialog*() и передать в качестве параметра идентификатор диалога (константу, которую надо объявить в коде программы).

Метод *dismissDialog*() прячет диалоговое окно (но не удаляет), не отображая его на экране. Окно остается в пуле диалоговых окон данной активности. При повторном отображении при помощи метода *showDialog*() будет использована кешированная версия окна.

Метод *removeDialog*() удаляет окно из пула окон данной активности. При повторном вызове метода *showDialog*() диалоговое окно придется создавать снова.

Рассмотрим пример создания диалогового окна на основе класса *Dialog*. Следует создать простейшую разметку для диалогового окна – текстовое поле внутри *LinearLayout*:

```
<?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:id="@+id/dialogTextView"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:clickable="true"
            android:textSize="22sp"/>

</LinearLayout>
```

В разметку главной активности можно добавить кнопку для вызова диалогового окна:

```
public void onClick(View v)
    // Выводим диалоговое окно на экран
    dialog.show();
}
```

В коде для главной активности нужно определить:

```
public class MainActivity extends FragmentActivity {
Dialog dialog;
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  ...

    dialog = new Dialog(MainActivity.this);
    // Устанавливаем заголовок
    dialog.setTitle("Dialog window");
    // Передаем ссылку на макет
    dialog.setContentView(R.layout.dialog);
    // Находим элемент TextView внутри  разметки
    // и устанавливаем ему соответствующий текст
    TextView text = (TextView)
        dialog.findViewById(R.id.dialogTextView);
    text.setText("Delete list?");
}
```

При запуске приложения на экране должен отобразиться диалог (рис. 5.12).

Рис. 5.12. Интерфейс с диалоговым окном

По умолчанию при показе диалогового окна главная активность затемняется. В документации есть константы, позволяющие управлять степенью затемнения:

```
WindowManager.LayoutParams lp = dialog.getWindow().getAttributes();
lp.dimAmount = 0.5f; // Уровень затемнения от 1.0 до 0.0
dialog.getWindow().setAttributes(lp);
```

Метод *onCreateDialog*() вызывается один раз при создании окна. После начального создания при каждом вызове метода *showDialog*()

будет срабатывать обработчик *onPrepareDialog*(). Переопределив этот метод, можно изменять диалоговое окно при каждом его выводе на экран. Это позволит привнести контекст в любое из отображаемых значений. Если требуется перед каждым вызовом диалогового окна изменять его свойства (например, текстовое сообщение или количество кнопок), то это можно реализовать внутри метода. В метод передаются идентификатор диалога и сам объект *Dialog*:

```
@TargetApi(Build.VERSION_CODES.N)
@Override
public void onPrepareDialog(int id, Dialog dialog) {

 switch(id) {
  case (ID_DIALOG) :
    SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss");
    Date currentTime = new Date(java.lang.System.currentTimeMillis());
    String dateString = sdf.format(currentTime);
    AlertDialog timeDialog = (AlertDialog)dialog;
    timeDialog.setMessage(dateString);
    break;
    }
}
```

Если в активности должны вызываться несколько различных диалоговых окон, следует определить целочисленный идентификатор для каждого диалога. Эти идентификаторы можно использовать в вызове метода *showDialog*() и в обработчике события *onCreateDialog*() в операторе *switch*:

```
static final int DIALOG_ID_2 = 0;
static final int DIALOG_ID_1 = 1;

protected Dialog onCreateDialog(int id) {

    Dialog dialog = null;
    switch(id) {
        case DIALOG_ID_1:
            // Код для работы
            break;
        case DIALOG_ID_2:
            // Код для работы с диалогом
            break;
        default:
            dialog = null;
    }
    return dialog;
}
```

5.2.2. Класс AlertDialog

Диалоговое окно *AlertDialog* является расширением класса *Dialog*, и это наиболее используемый класс диалоговых окон. В создаваемых

окнах можно задавать элементы: заголовок; текстовое сообщение; кнопки (от одной до трех); список; флажки; переключатели.

Рассмотрим пример создания *AlertDialog*. Сначала следует создать ресурс:

```
<string name="app_name">AlertDialogSimple</string>
<string name="exit">Выход</string>
<string name="save_data">Сохранить данные?</string>
<string name="yes">Да</string>
<string name="no">Нет</string>
<string name="cancel">Отмена</string>
<string name="saved">Сохранено</string>
```

Так как в приложении может использоваться несколько видов диалоговых окон, то нужно определить отдельный идентификатор *DIALOG_ ID_1*. Затем следует создать объект класса *AlertDialog.Builder*, передав в качестве параметра контекст приложения. Используя методы класса *Builder*, нужно задать для создаваемого диалога заголовок (метод *setTitle*()), текстовое сообщение в теле диалога (метод *setMessage*()), значок (метод *setIcon*()), а также кнопки через методы. Для отображения окна вызывается метод *show*():

```
public class MainActivity extends Activity {

    final int DIALOG_ ID_1 = 1;

      public void onclick(View v) {
        // Вызываем диалог
        showDialog(DIALOG_ID_1);
    }

      protected Dialog onCreateDialog(int id) {
        if (id == DIALOG_ID_1) {
            AlertDialog.Builder al = new AlertDialog.Builder(this);
            // Заголовок
            al.setTitle(R.string.exit);
            al.setMessage(R.string.save_data);
            al.setIcon(android.R.drawable.ic_dialog_info);
            al.setPositiveButton(R.string.yes, myClickListener);
            al.setNegativeButton(R.string.no, myClickListener);
            al.setNeutralButton(R.string.cancel, myClickListener);
      return al.create();
        }

        return super.onCreateDialog(id);
    }
```

На экране отображается следующий диалог (рис. 5.13):

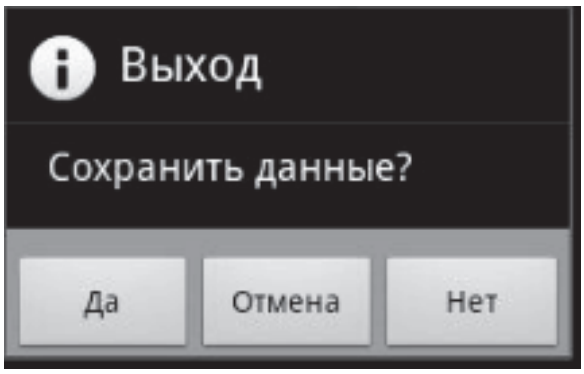


Рис. 5.13. Отображение диалога,
созданного на основе *AlertDialog*

Обработка нажатия кнопки определяется через параметр метода:

```
OnClickListener myClickListener = new OnClickListener() {

   public void onClick(DialogInterface dialog, int which) {

       switch (which) {
            // Положительная кнопка
            case Dialog.BUTTON_POSITIVE:
              break;
            // Негативная кнопка
            case Dialog.BUTTON_NEGATIVE:
              break;
            // Нейтральная кнопка
            case Dialog.BUTTON_NEUTRAL:
              break;
        }
    });
```

В *AlertDialog* можно добавить только по одной кнопке каждого типа: *Positive*, *Neutral* и *Negative*, т. е. максимально возможное количество кнопок в диалоге – три.

Диалоговое окно *AlertDialog* очень гибкое в настройках.

Чтобы диалоговые окна сохраняли состояние, рекомендуется использовать методы активности *onCreateDialog*() и *onPrepareDialog*().

5.2.3. Класс DialogFragment

Класс *DialogFragment* отображается как диалог и имеет соответствующие методы. Построить диалог можно двумя способами: используя *layout*-файл и через *AlertDialog.Builder*.

Сначала создается класс, например *DialogF.java* (рис. 5.14).

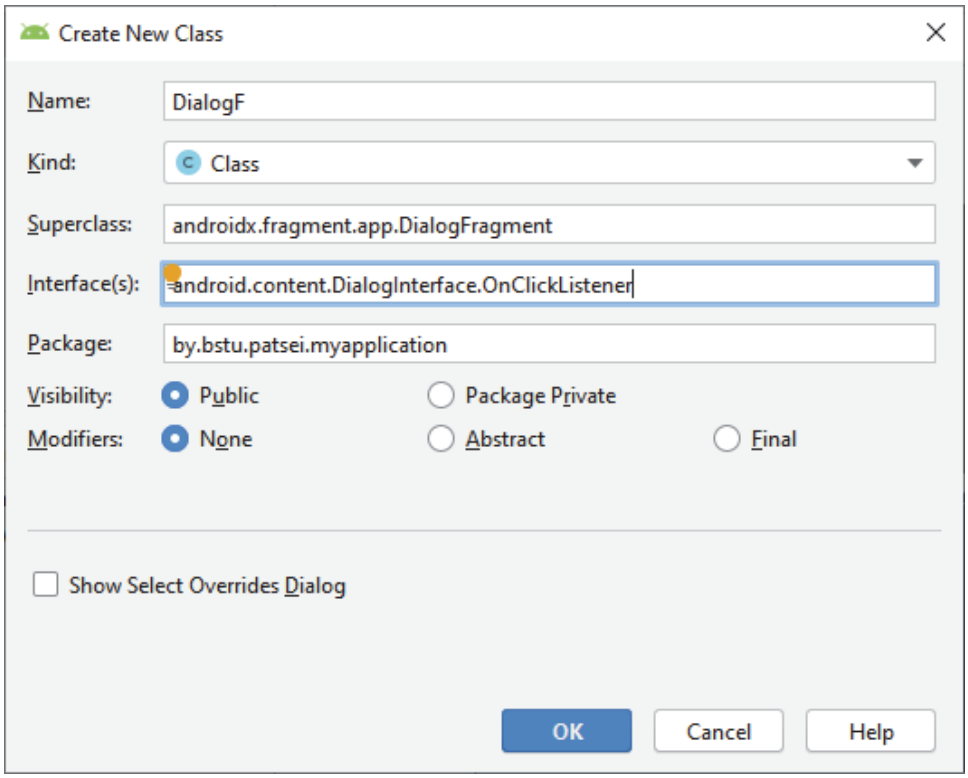


Рис. 5.14. Добавление класса диалога *DialogFragment*

Для создания диалога с помощью *Builder* используется метод обратного вызова *onCreateDialog*. Диалог будет иметь заголовок, сообщение и три кнопки. Обработчиком для кнопок назначается текущий фрагмент. С помощью *onClick* выводится соответствующий текст в лог. Диалог сам закроется по нажатии на кнопку, а метод *dismiss* здесь не нужен (методы *onDismiss* и *onCancel* – это закрытие и отмена диалога):

```
public class  DialogF  extends DialogFragment implements DialogInter-
face.OnClickListener {

      final String LOG_TAG = "myLogs";

       public Dialog onCreateDialog(Bundle savedInstanceState) {
            AlertDialog.Builder adb = new AlertDialog.Builder(getAc-
tivity())
           .setTitle("Title").setPositiveButton(R.string.yes, this)
           .setNegativeButton(R.string.no, this)
           .setNeutralButton(R.string.maybe, this)
           .setMessage(R.string.message_text);
            return adb.create();
        }
        public void onClick(DialogInterface dialog, int which) {
            int i = 0;
            switch (which) {
```

```
                case Dialog.BUTTON_POSITIVE:
                    i = R.string.yes;
                    break;
                case Dialog.BUTTON_NEGATIVE:
                    i = R.string.no;
                    break;
                case Dialog.BUTTON_NEUTRAL:
                    i = R.string.maybe;
                    break;
            }
            if (i > 0)
             Log.d(LOG_TAG, "Dialog Fragment: " +
getResources().getString(i));
        }

        public void onDismiss(DialogInterface dialog) {
            super.onDismiss(dialog);
            Log.d(LOG_TAG, "Dialog Fragment: onDismiss");
        }

        public void onCancel(DialogInterface dialog) {
            super.onCancel(dialog);
            Log.d(LOG_TAG, "Dialog Fragment: onCancel");
        }
    }
```

Определяется класс *MainActivity.java*, в котором диалог создается и запускается методом *show*, требующим на вход *FragmentManager* и строку-тег:

```
public class MainActivity extends AppCompatActivity {

    DialogFragment dlg;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dlg = new DialogF();
    }

    public void onclick(View v) {
          dlg.show(getSupportFragmentManager(),"dlg");
    }
}
```

На экране должно появиться следующее окно (рис. 5.15).

Есть еще один вариант вызова диалога. Это метод *show*(), но на вход он уже принимает не *FragmentManager*, а *FragmentTransaction*. В этом случае система сама вызовет *commit* внутри *show*, при этом можно предварительно поместить в созданную транзакцию еще какие-либо операции или отправить ее в *BackStack*.

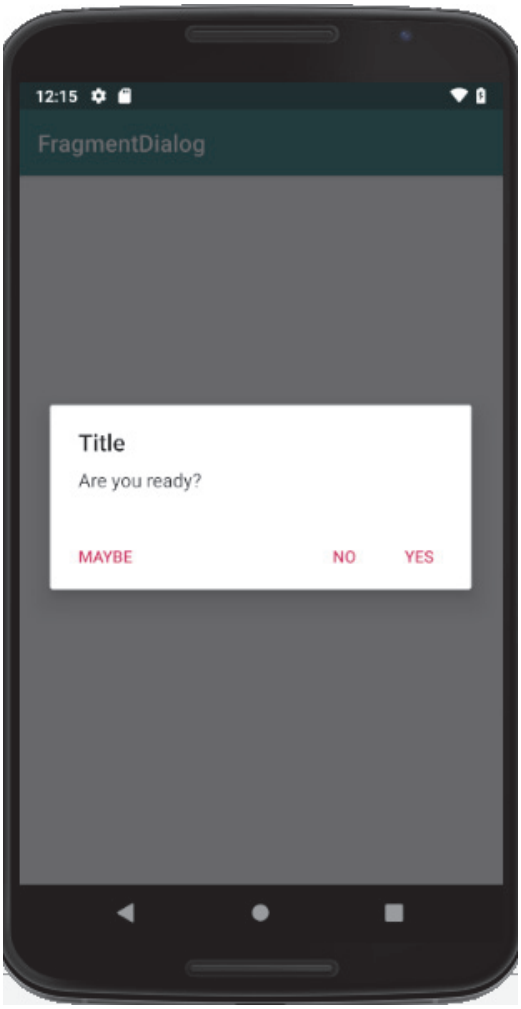


Рис. 5.15. Отображение диалога *DialogFragment*

Можно использовать диалог-фрагмент, как обычный фрагмент, и отображать его на *Activity*, а не в виде диалога.

5.2.4. Класс TimePickerDialog

Класс *TimePickerDialog* относится к стандартным диалогам, предоставляемым системой. Он позволяет указать время. Рассмотрим пример:

```
public class MainActivity extends AppCompatActivity {

    TimePickerDialog tpd;
    int myHour = 14;
    int myMinute = 35;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
tpd = new TimePickerDialog(this, myCallBack, myHour, myMinute, true);
```

```
    }

    TimePickerDialog.OnTimeSetListener myCallBack = new
    TimePickerDialog.OnTimeSetListener() {

    public void onTimeSet(TimePicker view, int hourOfDay, int minute)
       {
            myHour = hourOfDay;
            myMinute = minute;
            Toast.makeText(
               getApplicationContext(),
              "Time is " + myHour + " hours " + myMinute + " minutes",
               Toast.LENGTH_SHORT).show();
        }

    };

    public void onclick(View v) {
                        tpd.show();
        }
}
```

Также создается *TimePickerDialog* на основе следующего конструктора:

```
TimePickerDialog (Context context,
     TimePickerDialog.OnTimeSetListener callBack,
     int hourOfDay,
     int minute,
     boolean is24HourView)
```

Здесь *context* – контекст; *callback* – обработчик с интерфейсом *TimePickerDialog.OnTimeSetListener*, метод которого срабатывает при нажатии кнопки *OK* на диалоге; *hourOfDay* – час, который покажет диалог; *minute* – минута, которую покажет диалог; *is24HourView* – формат времени 24 часа (иначе AM/PM); *myCallBack* – объект, реализующий интерфейс *TimePickerDialog.OnTimeSetListener*.

В результате выполнения приложения на экране будет отображено следующее (рис. 5.16).

5.2.5. Класс DatePickerDialog

Этот класс считается устаревшим. Создать *DatePickerDialog* можно с использованием конструктора:

```
DatePickerDialog (Context context,
                  DatePickerDialog.OnDateSetListener callBack,
int year,
int monthOfYear,
int dayOfMonth)
```

Рис. 5.16. Отображение диалога *TimePickerDialog*

Здесь *context* – контекст; *callBack* – обработчик с интерфейсом *DatePickerDialog.OnDateSetListener*, метод которого срабатывает при нажатии кнопки *OK* на диалоге; *year* – год, который покажет диалог; *monthOfYear* – месяц; *dayOfMonth* – день; *myCallBack* – объект, реализующий интерфейс *DatePickerDialog.OnDateSetListener*, у которого только один метод – *onDateSet*. Например, рис. 5.17:

```
public class MainActivity extends AppCompatActivity {

    DatePickerDialog tpd;
    int myYear = 2019;
    int myMonth = 02;
    int myDay = 03;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        tpd = new DatePickerDialog(this, myCallBack, myYear, myMonth, myDay);
    }

   DatePickerDialog.OnDateSetListener myCallBack = new DatePicker-
Dialog.OnDateSetListener() {

   public void onDateSet(DatePicker view, int year, int monthOfYear,
                         int dayOfMonth) {
            myYear = year;
            myMonth = monthOfYear;
            myDay = dayOfMonth;
            Toast.makeText( getApplicationContext(),
                  "Today is " + myDay + "/" + myMonth + "/" + myYear,
                   Toast.LENGTH_SHORT).show();
        }
    };

   public void onclick(View v) {
        tpd.show();     }
}
```

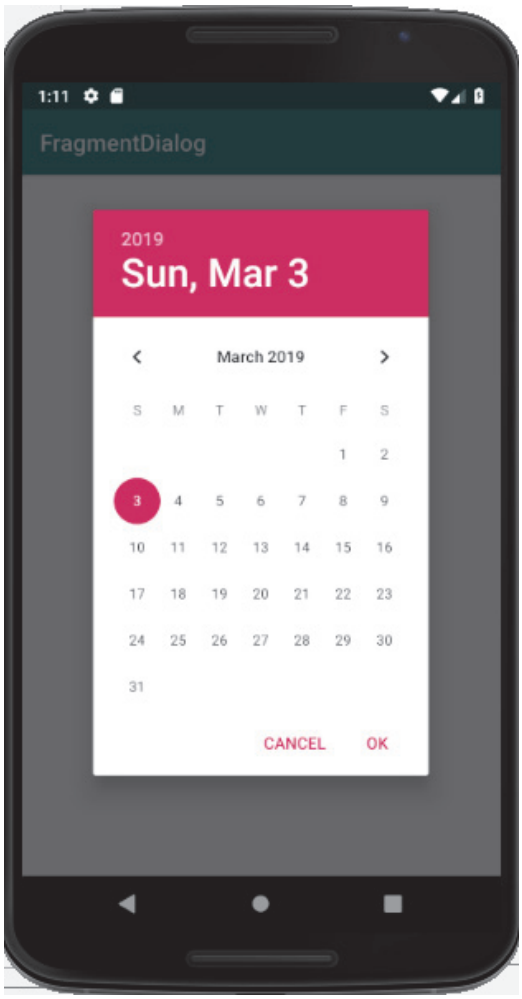


Рис. 5.17. Отображение диалога *DatePickerDialog*

5.2.6. Класс ShareActionProvider

Начиная с Android 4.0 (API 14) появился класс *android.widget.ShareActionProvider* – провайдер действия передачи информации, позволяющий реализовать интерфейс передачи данных другим приложениям. Это элементы меню с подходящими приложениями для обработки данных (рис. 5.18).

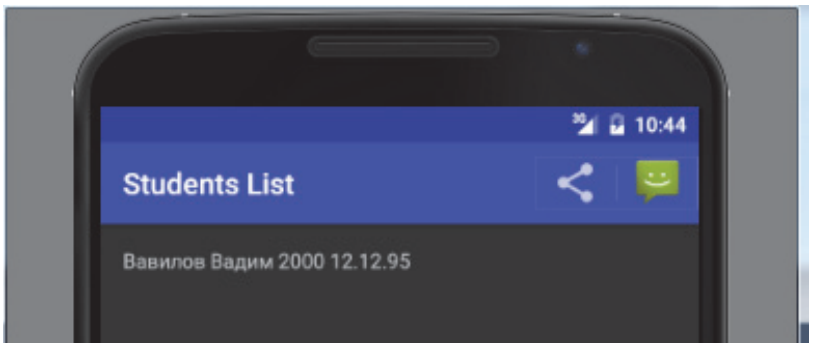


Рис. 5.18. Отображение *ShareActionProvider*

Если пользователь выберет подходящее приложение, то его значок можно сохранить для истории, чтобы в следующий раз можно было обойтись без вызова подменю.

6. НАВИГАЦИЯ В ACTIVITY И СПИСКОВЫЕ ПРЕДСТАВЛЕНИЯ

6.1. Навигация в Activity

6.1.1. Классификация Activity

Навигация в *Activity* связана с упорядочиванием и организацией взаимодействия *Activity*. Условно можно выделить три типа активностей: *активности верхнего уровня, активности категорий и активности детализации/редактирования.*

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

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

Активность детализации/редактирования выводит подробную информацию по конкретной записи, предоставляет пользователю возможность редактирования существующих записей или ввода новых значений. Пример *Activity* детализации/редактирования – активность, которая выводит подробную информацию о конкретном товаре, студенте, карте, письме и т. п.

Как правило, переход от активности верхнего уровня к активности детализации/редактирования должен осуществляться через *Activity* категорий.

6.1.2. Планирование нескольких размеров экрана

Приложениям Android необходимо адаптироваться к различным устройствам разных типов: от 3–10-дюймовых планшетов до 42-дюймовых телевизоров. 3–4-дюймовые экраны, как правило, подходят только для одновременного отображения одной вертикальной области содержимого, будь то список элементов или подробная информация об элементе. На таких устройствах экраны обычно отображают один

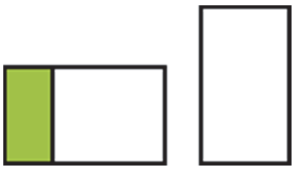
уровень в информационной иерархии (категория → список объектов → детализация объекта). Более крупные экраны имеют гораздо больше доступного пространства и могут представлять несколько панелей контента.

Многоуровневые макеты в ландшафтной ориентации работают хорошо из-за большого количества доступного горизонтального пространства. Однако в портретной ориентации горизонтальное пространство более ограничено, поэтому может потребоваться создать отдельный макет. Рассмотрим основные стратегии.

Растягивание. Самая простая стратегия состоит в том, чтобы растянуть ширину каждой панели для наилучшего представления содержимого. Панели могут иметь фиксированную ширину или принимать определенный процент от доступной ширины экрана.

Сворачивание – разворачивание. Позволяет динамически изменять ширину панели.

Показать – спрятать. В этом случае левая панель полностью скрыта в портретном режиме. Но она должна быть доступна через экранное изображение (например, кнопку). Обычно рекомендуется использовать кнопку *Вверх* в панели действий.



Стек. Стратегия состоит в том, чтобы вертикально складывать горизонтально расположенные панели в портрете.

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

Существует два типа активностей для сиблинга: связанные с коллекцией и связанные с разделом (рис. 6.1).

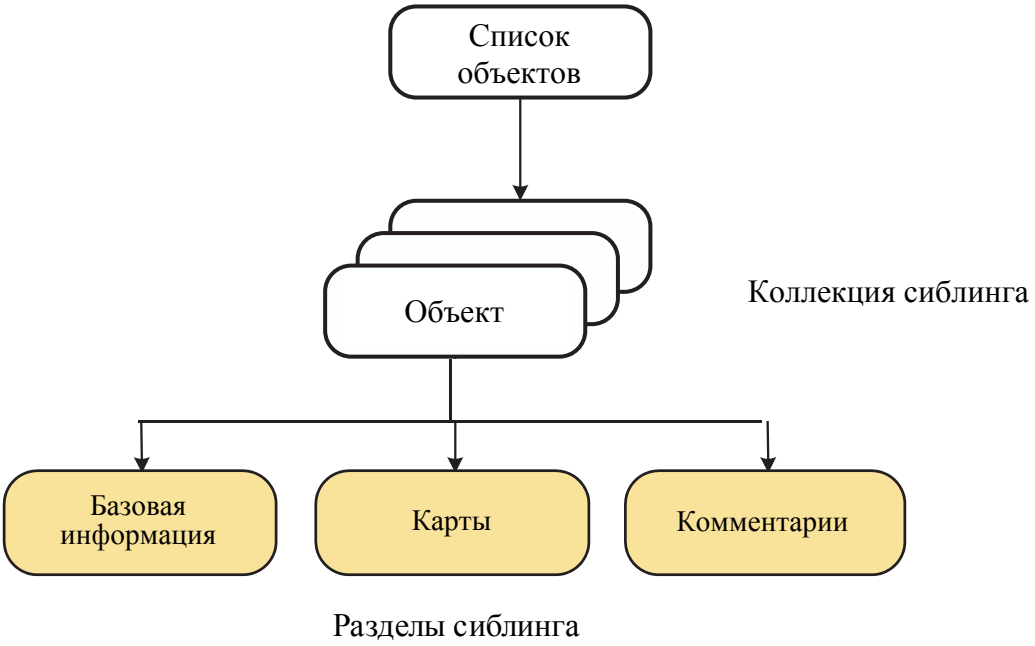


Рис. 6.1. Организация *Activity* с сиблингом

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

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

6.1.3. Шаблоны навигации

Навигация на основе кнопок. Простым шаблоном для доступа к различным разделам приложений верхнего уровня является шаблон на основе кнопок или панелей (рис. 6.2).

Списки, сетки и стеки. Для экранов с коллекциями, и особенно с текстовой информацией, списки с вертикальной прокруткой часто являются наиболее простым и знакомым видом интерфейса. Еще можно использовать фотографии или видео, вертикально прокручивающиеся

сетки элементов, списки горизонтальной прокрутки или стеки (называемые картами) (рис. 6.3).

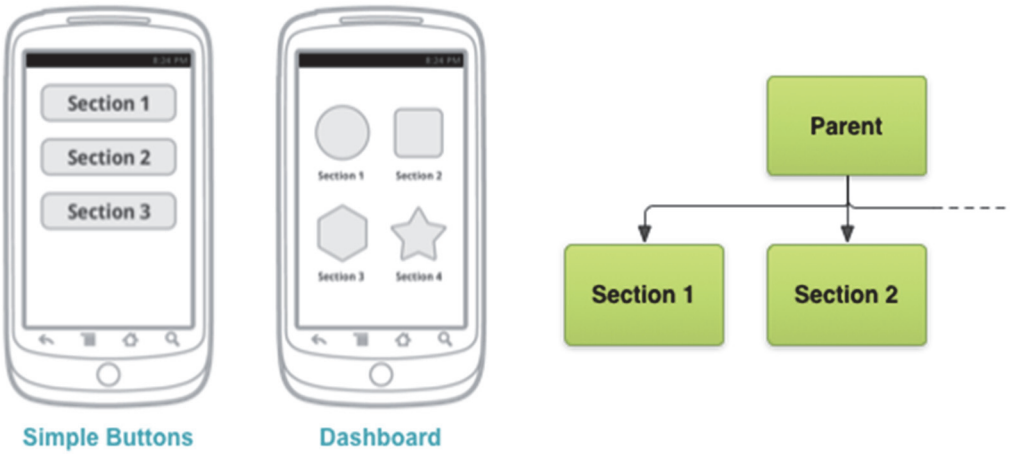


Рис. 6.2. Навигация *Activity* на основе кнопок

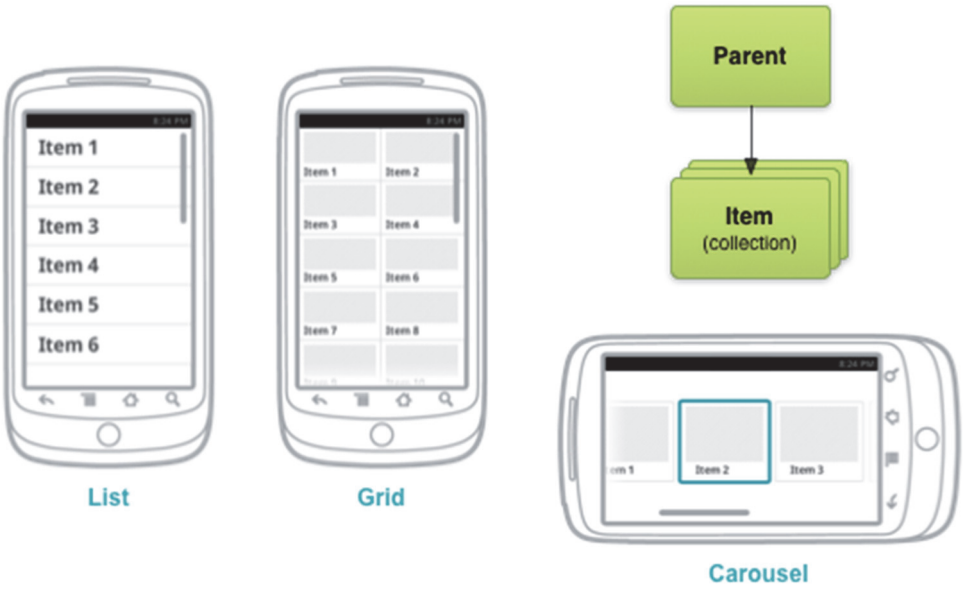


Рис. 6.3. Навигация *Activity* на основе списков и сеток

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

Навигация на основе вкладок. Использование вкладок (*Tabs*) – популярное решение для горизонтальной навигации. Вкладки больше подходят для небольших наборов (4 или меньше) экранов, связанных с разделом (рис. 6.4).

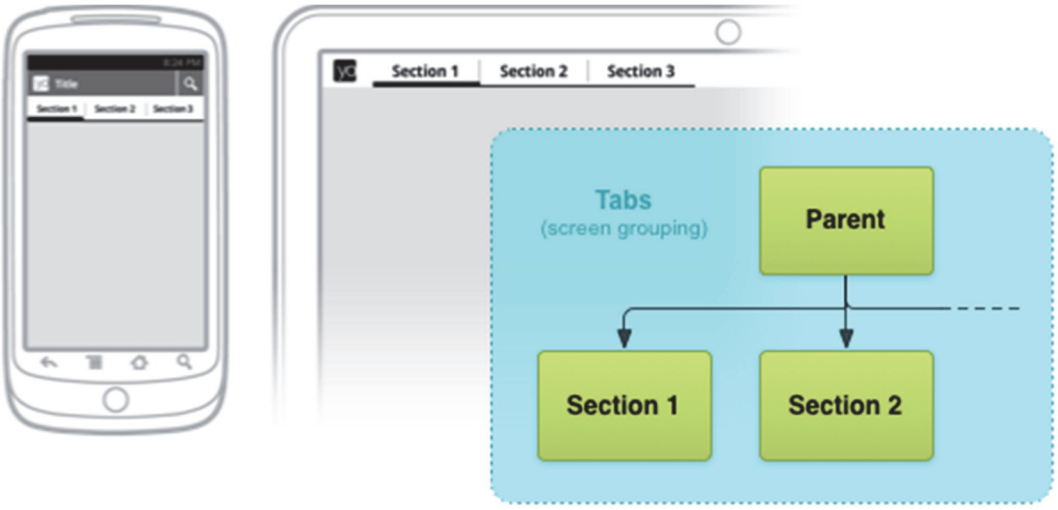


Рис. 6.4. Навигация *Activity* на основе *Tabs*

Существует несколько рекомендаций. Вкладки должны быть постоянными. Индикаторы вкладок должны оставаться доступными в любое время. Переключатели вкладок не должны рассматриваться как история. Например, если пользователь переключается с вкладки A на вкладку B, при нажатии кнопки *Назад* не следует переустанавливать вкладку A. Наконец, самое главное, вкладки должны всегда отображаться в верхней части экрана.

Навигация на основе горизонтальной прокрутки (*Swipe Views*). Другой популярной навигацией является горизонтальная прокрутка.

Использование вертикальных списков может также привести к неудобным взаимодействиям пользователей и плохому использованию места на больших экранах. Один из способов предоставить дополнительную информацию – разместить ее в отдельной горизонтальной панели, примыкающей к списку (рис. 6.5).

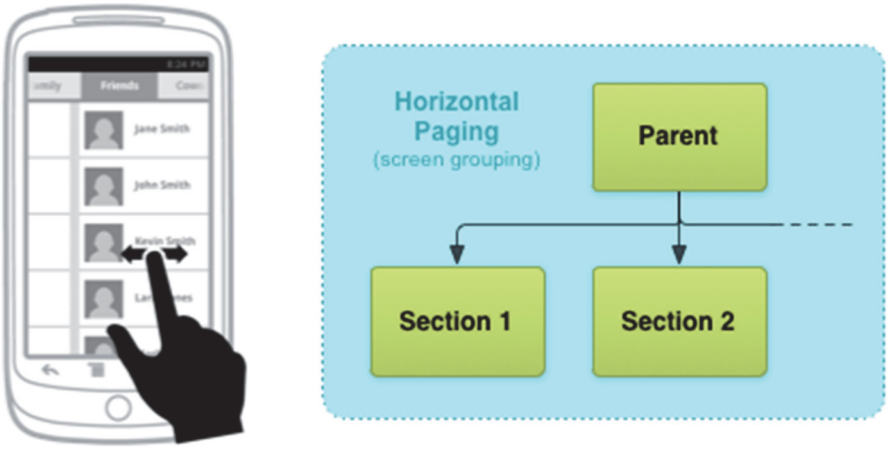


Рис. 6.5. Навигация *Activity* на основе *Swipe Views*

6.1.4. Навигация вперед – назад

Внутренне непротиворечивая навигация является важной составляющей пользовательского интерфейса.

Кнопка *Вверх* используется для навигации внутри приложения по иерархической структуре его экранов (рис. 6.6). Например, если на экране А отображается некоторый список и при выборе какого-либо элемента открывается экран В (с подробной информацией об этом элементе), то на экране В должна присутствовать кнопка *Вверх* для возврата к экрану А.

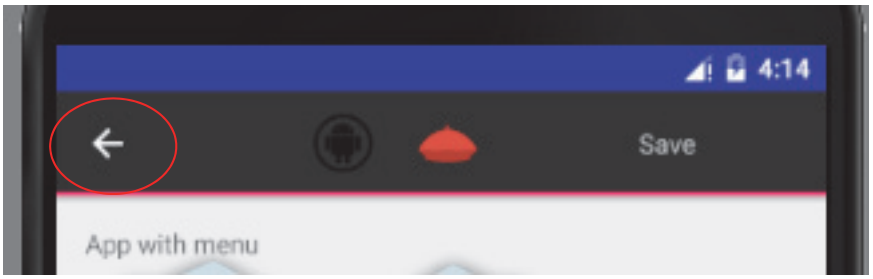


Рис. 6.6. Расположение кнопки *Вверх*

Если экран является самым верхним в приложении, он не должен содержать кнопку *Вверх*.

Системная кнопка *Назад* используется для навигации в обратном хронологическом порядке среди экранов, недавно открытых пользователем (рис. 6.7). Такая навигация основана на порядке появления экранов, а не на иерархии приложения.

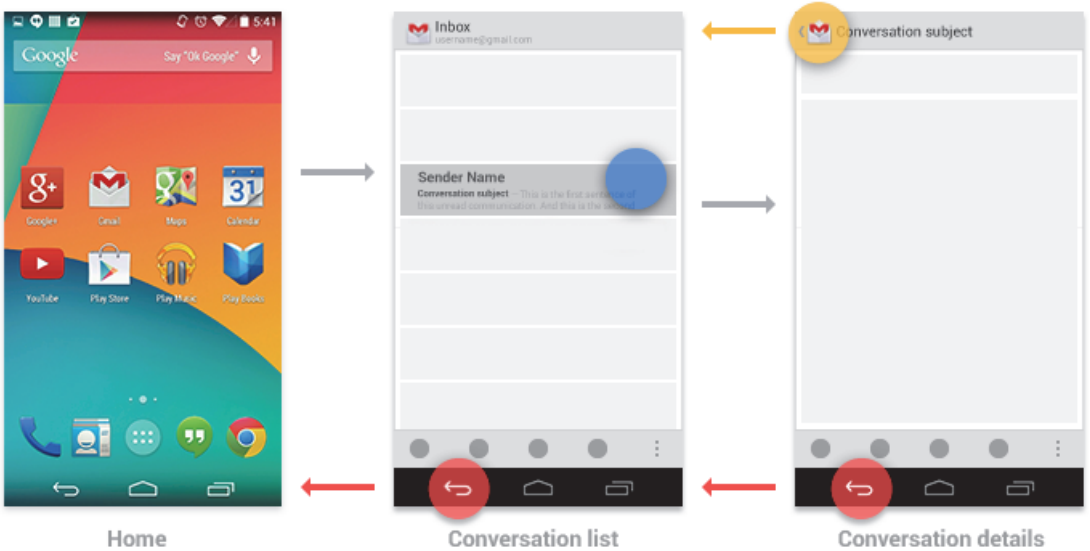


Рис. 6.7. Навигация в хронологическом порядке

Если предыдущий экран одновременно является иерархическим родителем текущего, кнопка *Назад* имеет то же действие, что и кнопка *Вверх* (рис. 6.7). Однако в отличие от кнопки *Вверх*, гарантирующей, что пользователь остается в приложении, кнопка *Назад* может перевести на главный экран или даже в другое приложение.

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

Навигация через виджеты приложений и уведомления. Например, виджет *Входящие* Gmail и новое уведомление о сообщениях могут обойти экран *Входящие*, подключая пользователя к просмотру сеанса.

Если экран назначения достигается с одного конкретного экрана в приложении, кнопка *Вверх* должна перейти к этому экрану. В противном случае кнопка *Вверх* должна перейти к самому верхнему экрану приложения (рис. 6.8).

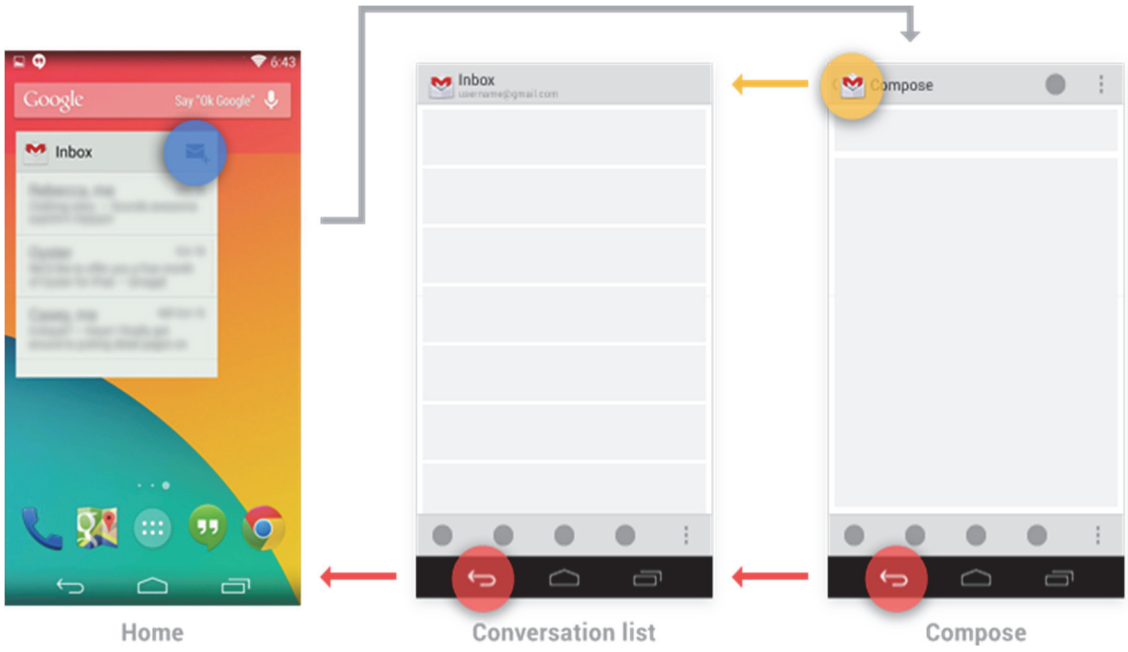


Рис. 6.8. Навигация через виджеты приложений и уведомления

Одним из основных преимуществ системы Android является способность приложений активировать друг друга, предоставляя пользователю возможность перемещаться напрямую из одного приложения в другое.

6.2. Навигация между Activity. Списковые Activity

6.2.1. Добавление кнопки Вверх

После создания первой активности, назначим ее родительской. Делается это в файле манифеста *AndroidManifest.xml*:

```
<activity

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

Ваше приложение должно облегчить пользователям поиск пути к главному экрану приложения. Простой способ сделать это – предоставить кнопку *Вверх* на панели для всех активностей, кроме основной. Когда пользователь выбирает кнопку *Вверх*, приложение переходит к родительской активности. Чтобы включить кнопку, необходимо вызвать метод *setDisplayHomeAsUpEnabled*(). Как правило, это делается, когда создается *Activity*. Метод *onCreate*() устанавливает панель инструментов и включает кнопку *Вверх* на панели приложений:

```
protected void onCreate(Bundle savedInstanceState) {

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

Если нет родителя, то ставим в параметрах *null*.

6.2.2. Добавление Activity для представления списка

В соответствии с описанием шаблонов навигации необходимо разработать шаблон, представленный на рис. 6.9.

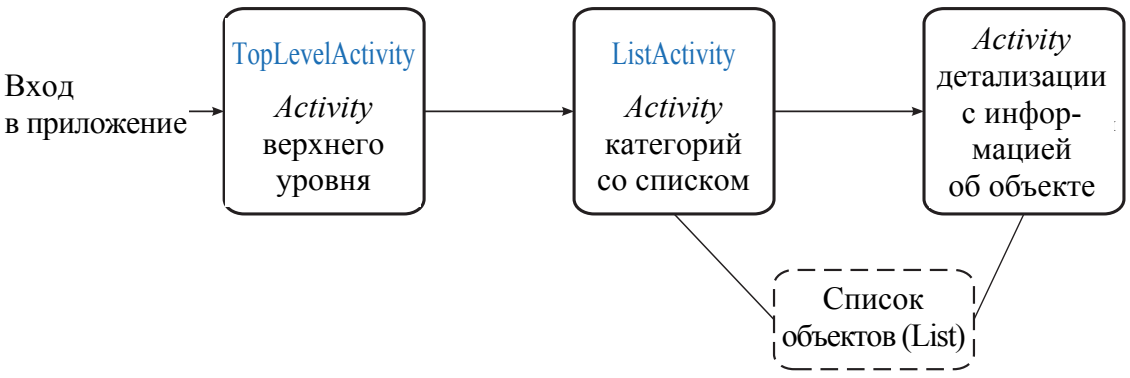


Рис. 6.9. Навигация между активностями в приложении

Объектом для отображения в списке будет студент. Определим список в виде динамического массива:

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

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

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

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

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

Списковое представление позволяет вывести вертикальный список объектов данных, который в дальнейшем может использоваться для навигации по приложению. Для добавления спискового представления в макете используется элемент *<ListView>*.

Чтобы заполнить списковое представление данными, используется атрибут *android:entries*, которому присваивается массив строк. Строки из массива будут отображаться в виде набора надписей *TextView*. Ниже приведен пример добавления в макет спискового представления, которое получает значения из массива строк:

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

Данные включаются в массив *strings.xml*:

```
…
<string-array name="options">

        <item>Students</item>
        <item>Listener</item>

</string-array>
```

В результате выводится список, как показано на рис. 6.10.

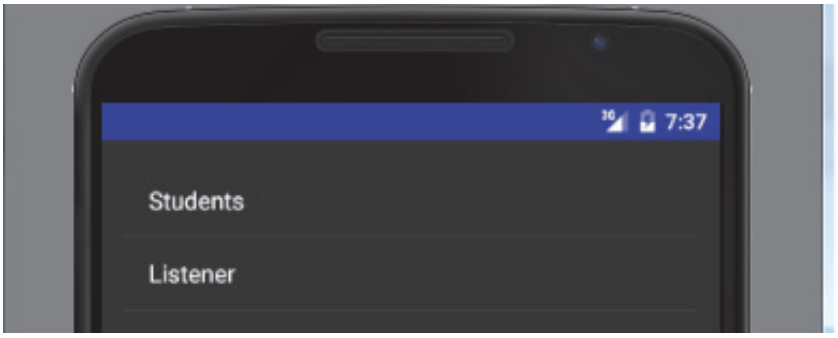


Рис. 6.10. Вывод списка на экран

Такой способ подходит только для данных, представленных статическим массивом.

Чтобы пункты списка реагировали на щелчки, следует реализовать слушателя событий. Реализация слушателя событий позволяет обнаруживать конкретные действия пользователя, например щелчки на вариантах списка, и реагировать на них. Для этого нужно создать объект типа *OnItemClickListener* и реализовать его метод *onItemClick*(). По параметрам, передаваемым методу *onItemClick*(), можно получить дополнительную информацию о событии, например получить ссылку на вариант из списка, узнать его позицию в списковом представлении (начиная с 0) и идентификатор записи используемого набора данных. В приведенном примере при щелчке на первом варианте спискового представления – варианте в позиции 0 – должна запускаться активность типа *StudentsCategoryActivity*. Необходимо создать интент для запуска *ListenerCategoryActivity*. Код создания слушателя выглядит так:

```
AdapterView.OnItemClickListener itemClickListener =
                          new AdapterView.OnItemClickListener(){

 public void onItemClick(AdapterView<?> listView, View itemView,
                         int position, long id) {

    if (position == 0) {
      Intent intent1 = new Intent(TopList.this,
                                  StudentsCategoryActivity.class);
                                  startActivity(intent1);
    }
    else {
    Intent intent2 = new Intent(TopList.this,
                                ListenerCategoryActivity.class);
    startActivity(intent2);
    }
    }
    }
```

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

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

Эта операция обеспечивает получение слушателем оповещений о том, что пользователь щелкает на списковом представлении.

Списковая активность специализируется на работе со списком. Она автоматически связывается со списковым представлением, поэтому не надо создавать такое представление самостоятельно (макет определяется на программном уровне). Макет, генерируемый списковой активностью, содержит одно представление. Для обращения к нему из кода активности используется метод *getListView*(). Класс *ListActivity* уже реализует слушателя событий, который обнаруживает щелчки на вариантах спискового представления. Поэтому достаточно реализовать метод *onListItemClick*() списковой активности.

Ниже приведен код создания списковой активности:

```
public class StudentsCategoryActivity extends ListActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

}
```

Этот код создает базовую списковую активность (он должен расширять класс *ListActivity* вместо класса *Activity*). Как уже упоминалось, не нужно назначать макет, используемый *ListActivity*, вызовом *setContentView*(). *ListActivity* должны быть зарегистрированы в файле *AndroidManifest.xml*.

Если списковое представление необходимо связать с данными, хранящимися в чем-то отличном от ресурса массива строк, придется написать код активности для привязки данных. В нашем примере списковое представление требуется связать с массивом *Students* из класса *Student*. Если данные спискового представления должны поступать из нестатического источника (например, из массива Java или базы данных), необходимо использовать адаптер. Адаптер играет роль моста между источником данных и списковым представлением (рис. 6.11).

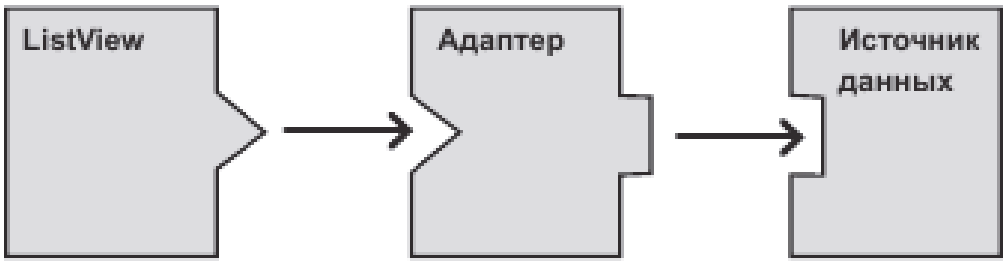


Рис. 6.11. Схема взаимодействия *ListView* и источника данных

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

В Android предусмотрено несколько подклассов адаптера. Наиболее часто используемые адаптеры: *ArrayAdapter* и *SimpleCursorAdapter*. *ArrayAdapter* используется в случае, когда в качестве источника данных выступает массив. По умолчанию *ArrayAdapter* создает представление для каждого элемента массива путем вызова метода *toString*() и помещения содержимого в объект *TextView*.

Чтобы использовать адаптер массива, следует проинициализировать его и присоединить к списковому представлению. При инициализации адаптера массива сначала указывается тип данных массива, *Context* (обычно текущей активности), ресурс макета (определяет, как должен отображаться каждый элемент из массива) и сам массив. Приведенный ниже листинг создает адаптер массива для отображения данных из массива студентов. Затем адаптер массива связывается со списковым представлением при помощи метода *setAdapter*() класса *ListView*:

```
ListView listStud = getListView();

ArrayAdapter<Student> listAdapter = new ArrayAdapter<Student>(
        this,
        android.R.layout.simple_list_item_1,
        Student.studList);

listStud.setAdapter(listAdapter);
```

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

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

StudentCategoryActivity создает *ArrayAdapter<Student>* – адаптер массива, содержащий объекты *Student*. Источником данных адаптера массива является массив *students* класса *Student*. *StudentCategoryActivity* заставляет *ListView* использовать адаптер массива, вызывая метод *setAdapter*() (рис. 6.12).

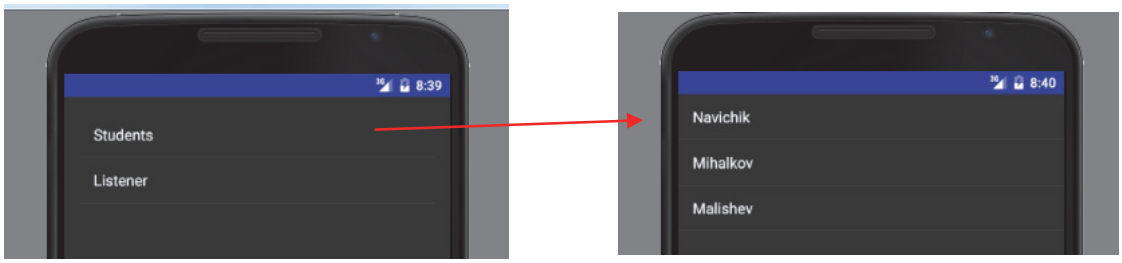


Рис. 6.12. Экраны *StudentCategoryActivity* и *StudentActivity*

Будем использовать *onListItemClick*() для запуска другой активности, которая выводит подробное описание варианта, выбранного пользователем. Для этого создается интент, открывающий активность. Идентификатор варианта, выбранного пользователем, включается в дополнительную информацию, чтобы новая активность могла использовать его при запуске. В нашем случае нужно запустить активность *StudentActivity* и передать ей идентификатор выбранного студента. *StudentActivity* использует эту информацию для вывода информации. Ниже приведен листинг:

```
public class StudentsCategoryActivity extends ListActivity {

    public static final String EXTRA_NUM = "-1";
…
@Override
public void onListItemClick(ListView listView,
                            View itemView,
                            int position,
                            long id) {
    Intent intent = new Intent (StudentsCategoryActivity.this,
                                StudentActivity.class);
    intent.putExtra(StudentyActivity.EXTRA_NUM, (int) id);
    startActivity(intent);

}
```

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

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

Добавим в проект новую активность с именем *StudentActivity* и макет с именем *activity_student*. Ниже приведен фрагмент разметки макета.

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

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

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

При запуске активность детализации читает из интента дополнительную информацию и использует ее для заполнения своих представлений. Информация из интента используется для получения данных, которые должны выводиться в подробном описании:

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

@Override
protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_student);

    //Получаем из данных интента
    int studNum = (Integer)getIntent().getExtras().get(EXTRA_NUM);
    Student st= Student.studList[studNum];
…
```

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

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

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

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

В результате получилось следующее (рис. 6.13). При запуске приложения открывается активность *TopListActivity*. Метод *onCreate*() активности *TopListActivity* создает объект *onItemClickListener* и связывает его с компонентом *ListView* активности.

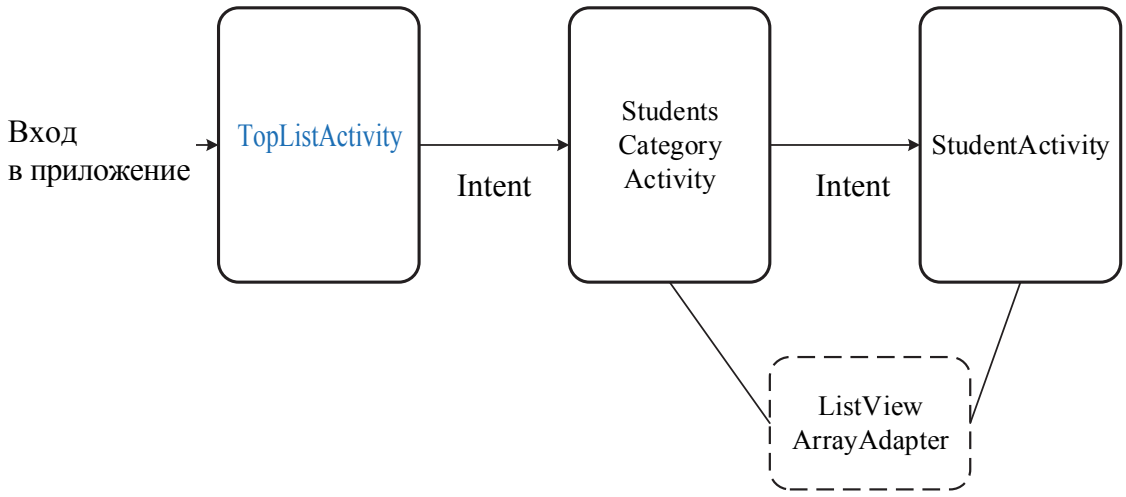


Рис. 6.13. Структура взаимодействия активностей

Когда пользователь щелкает на варианте спискового представления, вызывается метод *onItemClick*() объекта *onItemClickListener*. Если щелчок был сделан на варианте *Students*, то слушатель *onItemClickListener* создает интент для запуска *StudentCategoryActivity*.

StudentCategoryActivity наследуется от *ListActivity*. Списковое представление *StudentCategoryActivity* использует *ArrayAdapter*<> для вывода списка. Когда пользователь выбирает студента в *ListView*, вызывается метод *onListItemClick*(). Метод *onListItemClick*() класса *StudentCategoryActivity* создает интент для запуска *StudentActivity*, передавая номер в дополнительной информации.

Запускается активность *StudentActivity*. Активность читает номер из интента и получает подробную информацию о студенте из класса *Student*. Информация используется для обновления содержимого представлений (см. рис. 6.13).

6.2.3. Обзор классов адаптеров

Схему иерархии интерфейсов и классов адаптеров можно представить следующим образом (рис. 6.14): I – интерфейс; AC – абстрактный класс; C – класс; линии обозначают наследование.

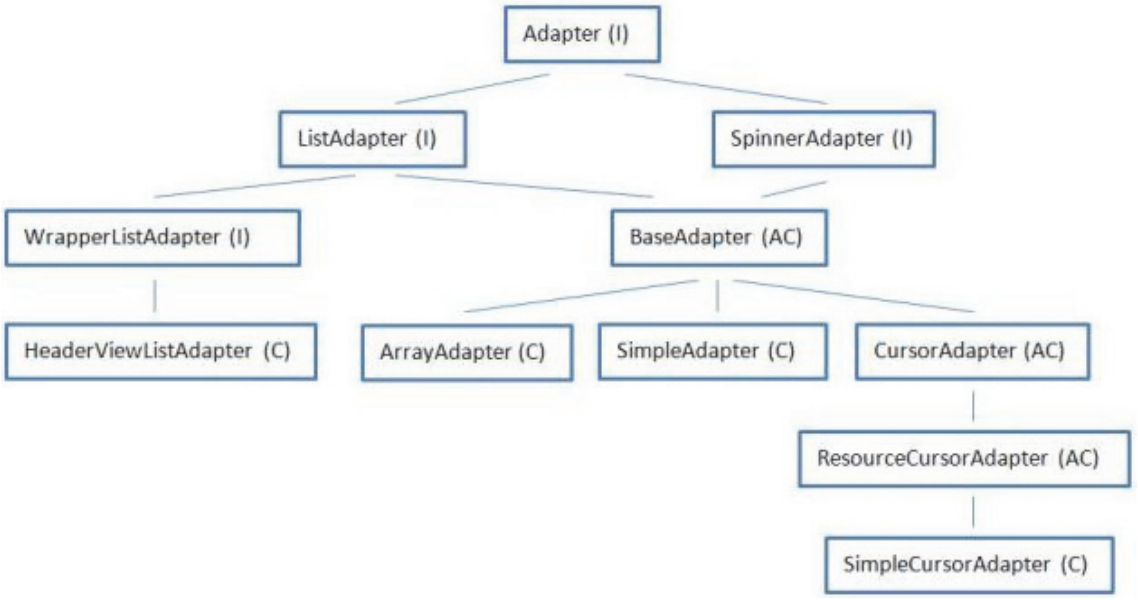


Рис. 6.14. Иерархия классов адаптеров

Интерфейс *Adapter* описывает базовые методы, которые должны содержать адаптеры: *getCount*, *getItem*, *getView* и др.

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

Интерфейс *SpinnerAdapter* используется адаптерами для построения *Spinner* (выпадающий список, или *drop-down*). Он содержит метод *getDropDownView*, который возвращает элемент выпадающего списка.

Интерфейс *WrapperListAdapter* используется для работы с вложенными адаптерами. Содержит метод *getWrappedAdapter*, который позволяет вытащить из основного адаптера вложенный.

Класс *HeaderViewListAdapter* – готовый адаптер для работы с *Header* и *Footer*. Внутри себя содержит еще один адаптер (*ListAdapter*).

Абстрактный класс *BaseAdapter* содержит немного своих методов и реализует методы интерфейсов, которые наследует, но не все. Своим наследникам оставляет на обязательную реализацию методы *getView*, *getItemId*, *getItem*, *getCount* из *ListAdapter*. Если нужно создать свой адаптер, этот класс подойдет.

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

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

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

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

Класс *SimpleCursorAdapter* – готовый адаптер, похож на *SimpleAdapter*, только использует не набор объектов *Map*, а *Cursor*, т. е. набор строк с полями. Соответственно, в массив *from*[] заносятся наименования полей, значения которых необходимо вытащить в соответствующие *View* из массива *to*.

Таким образом, есть четыре готовых адаптера: *HeaderViewListAdapter*, *ArrayAdapter*<T>, *SimpleAdapter*, *SimpleCursorAdapter*. Если имеется массив строк, то следует выбрать *ArrayAdapter*. Если работаем с базой данных и есть курсор, то используем *SimpleCursor-Adapter*.

Кроме рассмотренной иерархии есть адаптеры для работы с деревом – *ExpandableListView*. Здесь данные не одноуровневые, а делятся на группы и элементы.

7. ДОПОЛНИТЕЛЬНЫЕ ЭЛЕМЕНТЫ НАВИГАЦИИ

7.1. Navigation Drawer

Выдвижные панели (*Navigation Drawer*) – содержат ссылки на основные навигационные точки приложения (рис. 7.1).

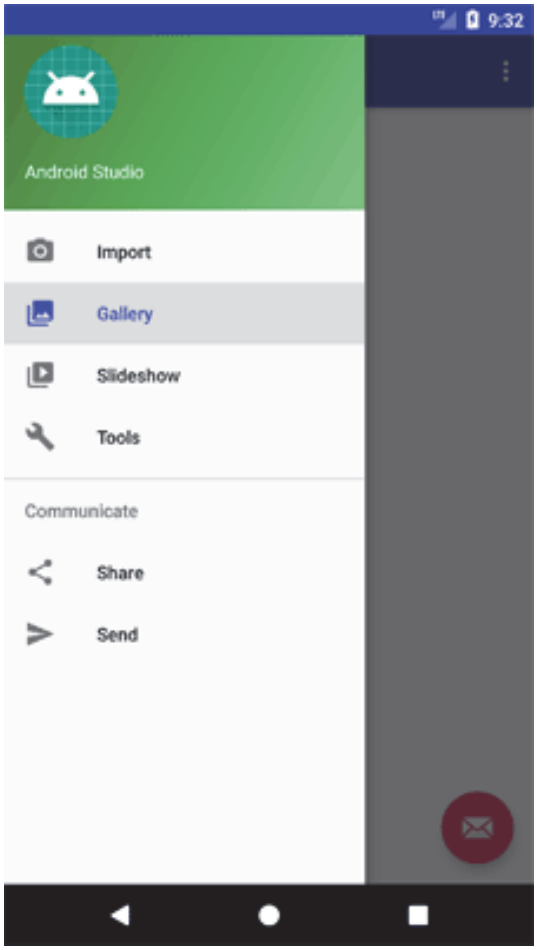


Рис. 7.1. Выдвижные панели *Navigation Drawer*

Navigation Drawer соответствует рекомендациям по дизайну (Google Material Design Guidelines – https://material.io/components/navigation-drawer/#anatomy).

Выдвижных панелей может быть несколько (слева и справа), они могут отображаться как под *StatusBar*, так и поверх него, позволяют менять иконки, цвета, бейджи во время выполнения (работают, начиная с API 14). При создании нового проекта можно выбрать нужный шаблон (рис. 7.2).

Для появления панели следует нажать на значок в виде трех горизонтальных полосок в заголовке. Значок в документации называется «гамбургером» (*Hamburger menu*). При нажатии слева появляется навигационная панель. По высоте она занимает весь экран, включая или не включая системную область в зависимости от версии операционной системы (рис. 7.3).

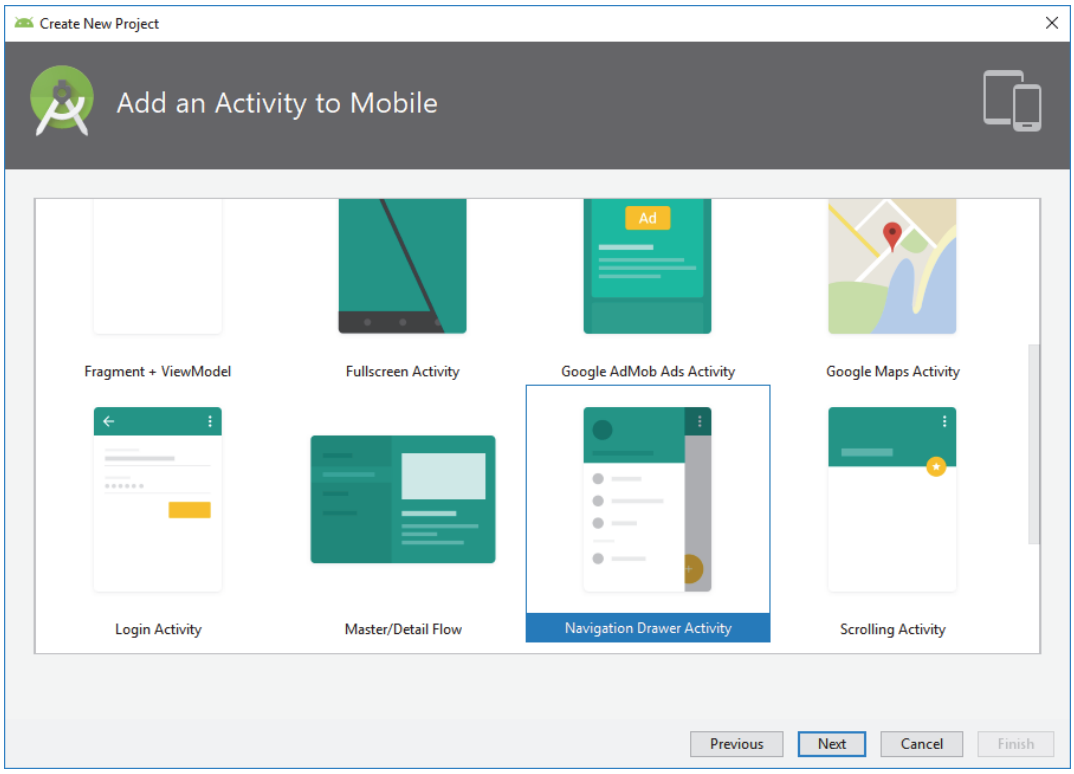


Рис. 7.2. Выбор шаблона

Сама панель состоит из двух основных частей. В верхней части находится картинка и текст, а в нижней – меню со значками. Меню, в свою очередь, разделено на две группы. В верхней части меню можно выбрать значки, и выбранный пункт останется выделенным. В нижней части меню пункты не выделяются.

Если открыть файл *activity_main.xml* в режиме *Design*, то можно увидеть, как будет выглядеть приложение с открытой панелью. Посмотрим на его содержание:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.drawerlayout.widget.DrawerLayout xmlns:android="http://sche-
mas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:openDrawer="start">

    <include
        layout="@layout/app_bar_main"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <com.google.android.material.navigation.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true"
        app:headerLayout="@layout/nav_header_main"
        app:menu="@menu/activity_main_drawer" />

</androidx.drawerlayout.widget.DrawerLayout>
```

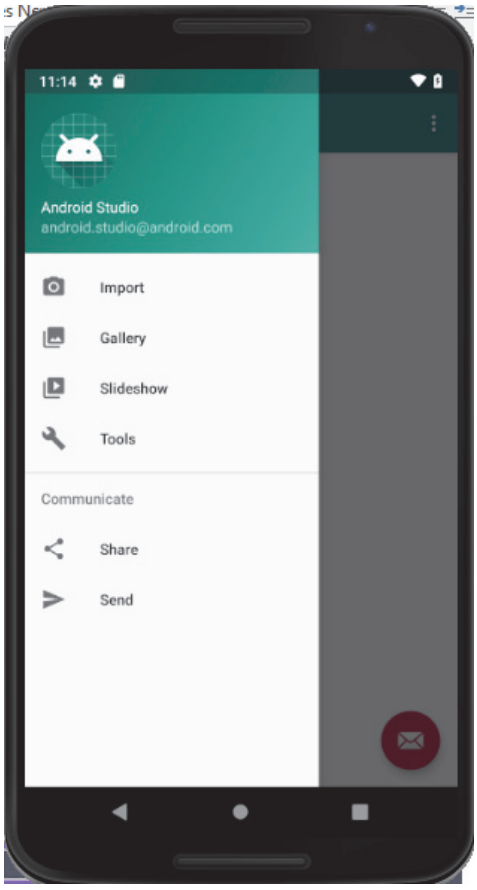


Рис. 7.3. Отображение *Navigation Drawer*

За выдвижную панель отвечает элемент *NavigationView*, который входит последним в контейнер *DrawerLayout* и представляет собой навигационное меню. А перед меню находится вставка *include*, указывающая на разметку *app_bar_main.xml*.

Атрибут *tools:openDrawer* позволяет указать, что навигационное меню нужно отобразить в раскрытом виде в режиме просмотра разметки.

Тег *NavigationView* содержит ссылку на собственную разметку в атрибуте *app:headerLayout*, который указывает на файл *nav_header_main.xml* (верхняя часть), а также на меню в атрибуте *app:menu*, который ссылается на ресурс меню *menu/activity_main_drawer.xml* (рис. 7.4).

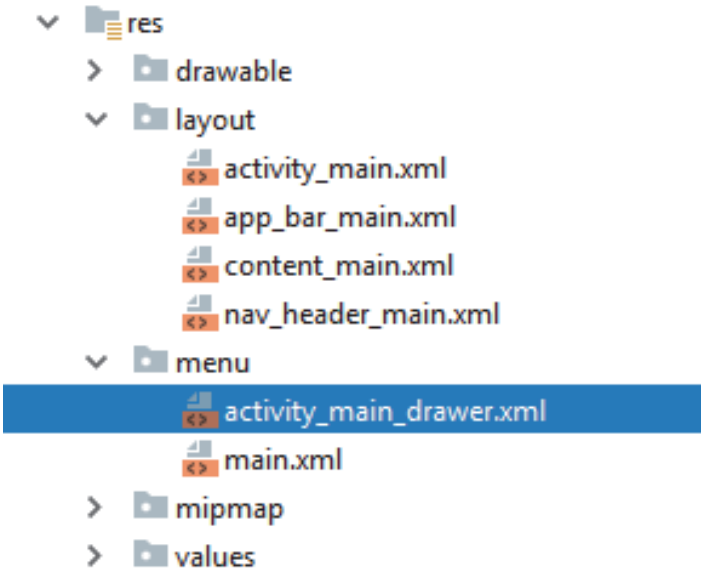


Рис. 7.4. Структура проекта

При открытии файла *nav_header_main.xml* можно увидеть следующую разметку:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/an-
droid"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="@dimen/nav_header_height"
    android:background="@drawable/side_nav_bar"
    android:gravity="bottom"
    android:orientation="vertical"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:theme="@style/ThemeOverlay.AppCompat.Dark">
```

```
    <ImageView
        android:id="@+id/imageView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:contentDescription="@string/nav_header_desc"
        android:paddingTop="@dimen/nav_header_vertical_spacing"
        app:srcCompat="@mipmap/ic_launcher_round" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/nav_header_vertical_spacing"
        android:text="@string/nav_header_title"
        android:textAppearance="@style/TextAppearance.AppCompat.Body1"
/>

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/nav_header_subtitle" />

</LinearLayout>
```

Разметка состоит из *ImageView* и двух *TextView*, размещенных в контейнере *LinearLayout*. Фон контейнера определен в ресурсе *drawable/side_nav_bar.xml* и представляет собой градиент.

Можно настроить верхнюю часть не через XML, а программно:

```
NavigationView navigationView = (NavigationView) findView-
ById(R.id.nav_view);

View headerLayout = navigationView.getHeaderView(0);
```

Теперь рассмотрим ресурс навигационного меню *res/menu/activity_main_drawer.xml*:

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

    <group android:checkableBehavior="single">

        <item
            android:id="@+id/nav_camera"
            android:icon="@drawable/ic_menu_camera"
            android:title="Import" />

        <item
            android:id="@+id/nav_gallery"
            android:icon="@drawable/ic_menu_gallery"
            android:title="Gallery" />
```

```
        <item
            android:id="@+id/nav_slideshow"
            android:icon="@drawable/ic_menu_slideshow"
            android:title="Slideshow" />

        <item
            android:id="@+id/nav_manage"
            android:icon="@drawable/ic_menu_manage"
            android:title="Tools" />
    </group>

    <item android:title="Communicate">
        <menu>

            <item
                android:id="@+id/nav_share"
                android:icon="@drawable/ic_menu_share"
                android:title="Share" />

            <item
                android:id="@+id/nav_send"
                android:icon="@drawable/ic_menu_send"
                android:title="Send" />

        </menu>
    </item>

</menu>
```

Затем изучим код активности для работы с панелью. В классе активности реализуется интерфейс *OnNavigationItemSelectedListener* с его методом *onNavigationItemSelected*().

Принцип создания элементов меню стандартный. Каждый пункт меню представляет собой тег <*item*> с указанием значка и текста. Для группировки используется элемент <*group*>. Поведение элементов меню в группе регулируется атрибутом *android*:*checkableBehavior*. В примере используется значение *single* – при нажатии на пункт меню он останется выделенным (принцип переключателя *RadioButton*). Всего доступно три варианта: *single* – можно выбрать один элемент группы (переключатель); *all* – можно выбрать все элементы группы (флажок); *none* – элементы не выбираются.

Логика ничем не отличается от работы с обычным меню. Определяется идентификатор выбранного пункта и нужно написать свой код. Затем вызывается метод *closeDrawer*() для закрытия панели:

```
public class MainActivity extends AppCompatActivity
        implements NavigationView.OnNavigationItemSelectedListener {
...
    @SuppressWarnings("StatementWithEmptyBody")
    @Override
```

```
    public boolean onNavigationItemSelected(MenuItem item) {
        // Handle navigation view item clicks here.
        int id = item.getItemId();
        if (id == R.id.nav_camera) {
            // Handle the camera action
        } else if (id == R.id.nav_gallery) {
        } else if (id == R.id.nav_slideshow) {
        } else if (id == R.id.nav_manage) {
        } else if (id == R.id.nav_share) {
        } else if (id == R.id.nav_send) {
        }

DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
drawer.closeDrawer(GravityCompat.START);
return true;
    }
}
```

В методе *onCreate*() выполняется инициализация:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Toolbar toolbar = findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
    FloatingActionButton fab = findViewById(R.id.fab);
    fab.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Snackbar.make(view, "Replace with your own action", Snack-
bar.LENGTH_LONG)
                    .setAction("Action", null).show();
        }
    });
    DrawerLayout drawer = findViewById(R.id.drawer_layout);
    NavigationView navigationView = findViewById(R.id.nav_view);
    // Passing each menu ID as a set of Ids because each
    // menu should be considered as top level destinations.
    mAppBarConfiguration = new AppBarConfiguration.Builder(
            R.id.nav_home, R.id.nav_gallery, R.id.nav_slideshow,
            R.id.nav_tools, R.id.nav_share, R.id.nav_send)
            .setDrawerLayout(drawer)
            .build();

    NavController navController = Navigation.findNavController(this,
R.id.nav_host_fragment);
    NavigationUI.setupActionBarWithNavController(this, navController,
mAppBarConfiguration);
    NavigationUI.setupWithNavController(navigationView, navControl-
ler);
}
```

При нажатии кнопки *Назад* проверяется состояние панели. Если она открыта (*isDrawerOpen*()), то следует закрыть ее с помощью метода *closeDrawer*():

```
@Override
public void onBackPressed() {

 DrawerLayout drawer = (DrawerLayout)findViewById(R.id.drawer_layout);
    if (drawer.isDrawerOpen(GravityCompat.START)) {
        drawer.closeDrawer(GravityCompat.START);
    } else {
        super.onBackPressed();
    }

}
```

Если надо выдвинуть панель справа, то устанавливается значение *end* у атрибута *layout_gravity* и изменяется макет:

```
<android.support.v7.widget.DrawerLayout
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        />
    <ListView
        android:layout_width="200dp"
        android:layout_height="match_parent"
        />

</android.support.v7.widget.DrawerLayout>
```

Добавляются зависимости:

```
dependencies { compile 'com.android.support:appcompat-v7:23.1.0' }
```

Определяются свойства *Drawer*:

```
<ListView android:id="@+id/drawer"
    android:layout_width="200dp"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    android:choiceMode="singleChoice"
    android:divider="@android:color/transparent"
    android:dividerHeight="0dp"
    android:background="#ffffff"/>
/>
```

Затем определяется макет для отдельного элемента списка в *res/layout/drawer_list_item.xml*:

```
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/text1"
    android:layout_width="match_parent"
```

```
    android:layout_height="wrap_content"
    android:background="?android:attr/activatedBackgroundIndicator"
    android:gravity="center_vertical"
    android:minHeight="?android:attr/listPreferredItemHeightSmall"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:textAppearance="?android:attr/textAppearanceListItemSmall"
    android:textColor="#fff" />
```

Инициализируется список:

```
public class TopActivity extends AppCompatActivity {

    private String[] drawer;
    private ListView drawerList;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_top);
        //--------------------
        drawer= getResources().getStringArray(R.array.drawer);
        drawerList = (ListView)findViewById(R.id.drawer);
        drawerList.setAdapter(new ArrayAdapter<String>(this,
              android.R.layout.simple_list_item_activated_1,drawer));
        drawerList
          .setOnItemClickListener(new DrawerItemClickListener());
```

Задается массив с элементами:

```
<string-array name="drawer">

    <item>Top</item>
    <item>List Student</item>
    <item>Help</item>

 </string-array>
```

Для обработки щелчков определяется слушатель для элементов списка в выдвижной панели:

```
private class DrawerItemClickListener implements
                              ListView.OnItemClickListener {
    @Override
    public void onItemClick(AdapterView<?> parent, View view,
                                    int position, long id){
        selectItem(position);
        //Код, выполняемый при щелчке на элементе списка.
    }
};
…
private void selectItem(int position) {
```

```
Fragment fragment = new Fragment();
        switch(position) {
            case 1:
                fragment = new TopFragment();
                break;
            case 2:
                fragment = new StudentListFragment();
                break;
            case 3:
                fragment = new HelpFragment();
                break;
            default:

        }
     FragmentTransaction ft = getFragmentManager().beginTransaction();
     ft.replace(R.id.fragment_top_drawer, fragment);
     ft.addToBackStack(null);
     ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
     ft.commit();

     DrawerLayout drawerLayout = (DrawerLayout) findView-
ById(R.id.drawer_layout);
     drawerLayout.closeDrawer(drawerList);
};
```

Запускается приложение. Выдвижная панель содержит три пункта (рис. 7.5).

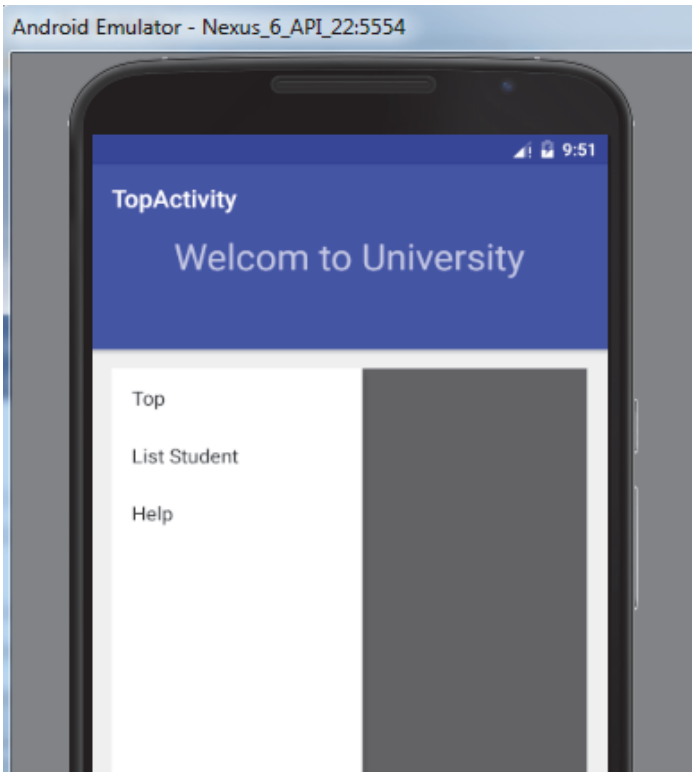


Рис. 7.5. Экран с выдвижной панелью

7.2. TabHost и TabWidget

TabHost и *TabWidget* – это вкладки, которые помогают логически разделить содержимое экрана. Вместо того чтобы переходить по разным экранам, можно сделать вкладки и переключаться между ними.

TabHost – корневой элемент вкладок. В нем определен вертикальный *LinearLayout*, в котором расположены *TabWidget* и *FrameLayout*. *TabWidget* будет отображать заголовки вкладок, а *FrameLayout* – содержимое. Во *FrameLayout* размещаются все *View*-компоненты, которые необходимо отобразить на вкладках.

```
<TabHost
    android:id="@+id/tabHost"
    android:layout_gravity="center_horizontal">

    <LinearLayout
      android:orientation="vertical">

        <TabWidget
            android:id="@android:id/tabs"
        </TabWidget>

        <FrameLayout
            android:id="@android:id/tabcontent">
            <LinearLayout
                android:id="@+id/linearLayout"
           android:orientation="vertical">
                <TextView
                    android:id="@+id/text1"
               android:text="Список студентов" />
            </LinearLayout>

            <LinearLayout
                android:id="@+id/linearLayout2"
           android:orientation="vertical">
                <TextView
                    android:id="@+id/text2"
              android:text="Список слушателей" />
            </LinearLayout>

        </FrameLayout>
    </LinearLayout>
</TabHost>
```

Позже вкладке сообщается, какой именно компонент она должна показать (явно указывается *id*), вкладка выберет нужный ей компонент и отобразит его как свое содержимое.

Разрабатывается интерфейс, который показан на рис. 7.6.

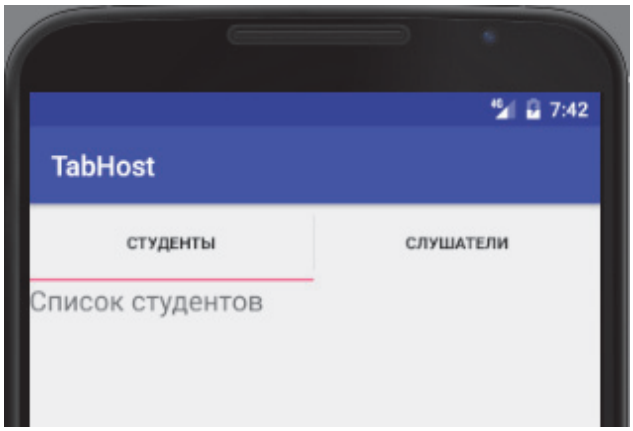


Рис. 7.6. Экран с *TabHost* с *TabWidget*

Находится компонент *TabHost*. Далее создаются две вкладки. Для создания используется метод *newTabSpec*, на вход он берет тег (некий строковый идентификатор вкладки). Для первой вкладки задается название методом *setIndicator*(). В метод *setContent*() передается *id* компонента (из *FrameLayout*), который должен выступать в качестве содержимого вкладки. В данном случае это *TextView*. Метод *addTab* присоединяет готовую вкладку к *TabHost*.

Вторая вкладка создается аналогично, только используется другая реализация метода *setIndicator*(). Заголовок вкладки может содержать не только текст, но и картинку. И здесь в метод передаются текст и XML вместо картинки.

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

    setTitle("TabHost");

    TabHost tabHost = (TabHost) findViewById(R.id.tabHost);
    tabHost.setup();

    TabHost.TabSpec tabSpec = tabHost.newTabSpec("tag1");
    tabSpec.setContent(R.id.linearLayout);
    tabSpec.setIndicator("Студенты");
    tabHost.addTab(tabSpec);

    tabSpec = tabHost.newTabSpec("tag2");
    tabSpec.setContent(R.id.linearLayout2);
    tabSpec.setIndicator("Слушатели", getResources().
                getDrawable(R.drawable.ptt));
    tabHost.addTab(tabSpec);

    tabHost.setCurrentTab(0);
}
```

В качестве вкладки можно использовать *Activity*. Основную активность, которая содержит *TabHost*, нужно наследовать не от *Activity*, а от *android.app.TabActivity*:

```
public class MainActivity extends TabActivity {

// Получаем TabHost
TabHost tabHost = getTabHost();
tabSpec = tabHost.newTabSpec("tag1");
tabSpec.setIndicator("Студенты");

tabSpec.setContent(new Intent(this, StudentActivity.class));
tabHost.addTab(tabSpec);
```

7.3. ViewPager

Часто приложения реализуют систему перелистывания, то есть приложение представляет собой набор страниц, которые можно перелистывать влево и вправо. В приложении Android для создания подобного эффекта можно использовать элемент *ViewPager*.

ViewPager отвечает за показ и прокрутку. Но ему нужен еще *PagerAdapter*, который отвечает за предоставление данных.

PagerAdapter – это базовый абстрактный класс, для которого разработчик дописывает реализацию так, как ему надо.

Существует распространенная стандартная (частичная) реализация *PagerAdapter*, которая работает с фрагментами *FragmentPager-Adapter*.

Android SDK содержит две встроенные реализации *PagerAdapter*: классы *FragmentPagerAdapter* и *FragmentStatePagerAdapter*.

Однако оба класса являются абстрактными, поэтому напрямую использовать их нельзя и нужно создавать класс-наследник.

FragmentStatePagerAdapter – быстрый адаптер, не требует пересоздания, но затратный, так как все держит в памяти (рис. 7.7). Подходит для небольшого количества страниц. Например, набор вкладок или визард.

Сами фрагменты не уничтожаются, но уничтожается их *View*-структура и потом создается заново. Хранится структура только текущей страницы и по одной справа и слева. Количество соседних страниц с сохраняемой *View*-структурой может быть настроено методом *setOffscreenPageLimit*.

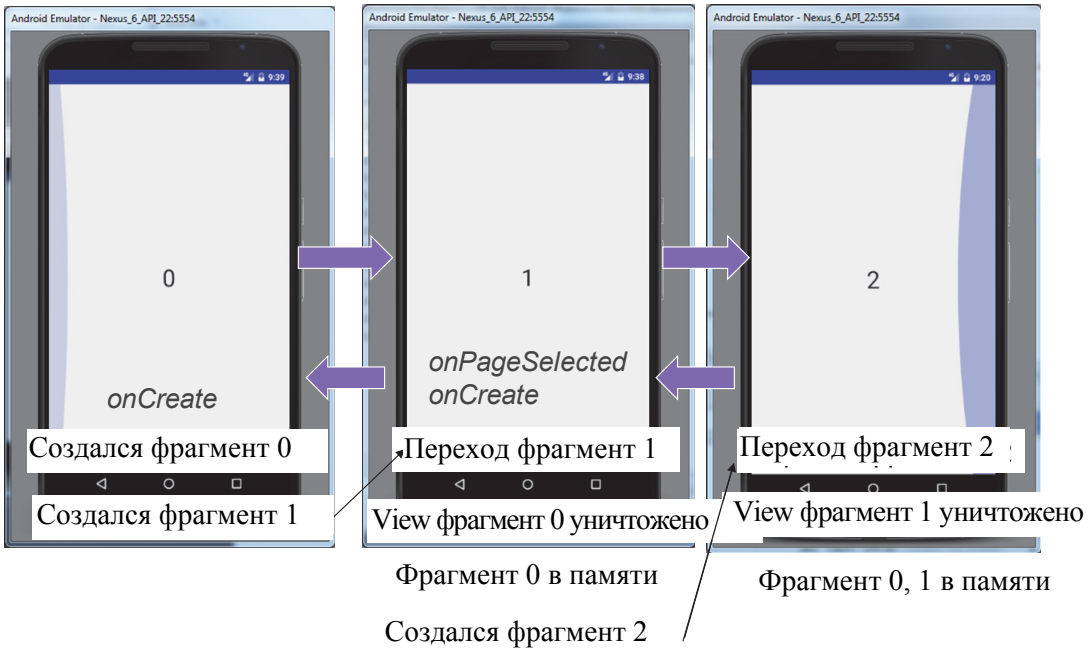


Рис. 7.7. Схема создания и уничтожения фрагментов

Сначала создается основной макет:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:an-
droid="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.viewpager2.widget.ViewPager2
        android:id="@+id/view_pager"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        />


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

Затем макет фрагмента. Здесь будет только *TextView*, который показывает содержимое страницы:

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

```
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".CardFragment"
    >
    <TextView
        android:id="@+id/tv_counter"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:textColor="#fff"
        android:textSize="20sp"
        android:textStyle="bold"
        tools:text="blank frgamnet"
        />
</FrameLayout>
```

Добавляются цвета в файл *color.xml:*

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
        <color name="colorPrimary">#008577</color>
        <color name="colorPrimaryDark">#00574B</color>
        <color name="colorAccent">#D81B60</color>
        <color name="red_100">#F44336</color>
        <color name="red_300">#3F51B5</color>
        <color name="red_500">#9C27B0</color>
        <color name="red_700">#2196F3</color>
        <color name="blue_100">#00BCD4</color>
        <color name="blue_300">#7BAAF7</color>
        <color name="blue_500">#009688</color>
        <color name="blue_700">#3367D6</color>
        <color name="green_100">#CDDC39</color>
        <color name="green_300">#57BB8A</color>
        <color name="green_500">#673AB7</color>
        <color name="green_700">#0B8043</color>

</resources>
```

Также надо создать класс фрагмента. Метод *newInstance*() создает новый экземпляр фрагмента и записывает ему в атрибуты число, которое пришло на вход. Это число – номер страницы для показа *ViewPager*. По нему фрагмент будет определять, какое содержимое создавать. В *onCreate* следует прочитать номер страницы из аргументов, в *onCreateView* – создать *View*, найти на нем *TextView*, написать простой текст с номером страницы.

```
import android.view.View;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.ViewGroup;
import android.widget.TextView;
import androidx.annotation.NonNull;
```

```
import androidx.annotation.Nullable;
import androidx.core.content.ContextCompat;
import androidx.fragment.app.Fragment;

public class CardFragment extends Fragment {
    private static final String ARG_COUNT = "param1";
    private Integer counter;
    private int[] COLOR_MAP = {
            R.color.red_100, R.color.red_300, R.color.red_500,
R.color.red_700, R.color.blue_100,
            R.color.blue_300, R.color.blue_500, R.color.blue_700,
R.color.green_100, R.color.green_300,
            R.color.green_500, R.color.green_700
    };
    public CardFragment() {
        // Required empty public constructor
    }
    public static CardFragment newInstance(Integer counter) {
        CardFragment fragment = new CardFragment();
        Bundle args = new Bundle();
        args.putInt(ARG_COUNT, counter);
        fragment.setArguments(args);
        return fragment;
    }
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (getArguments() != null) {
            counter = getArguments().getInt(ARG_COUNT);
        }
    }
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.card_fragment, container, false);
    }
    @Override public void onViewCreated(@NonNull View view, @Nullable
Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        view.setBackgroundColor(ContextCompat.getColor(getContext(),
COLOR_MAP[counter]));
        TextView textViewCounter = view.findViewById(R.id.tv_counter);
        textViewCounter.setText("Fragment No " + counter);
    }
}
```

Создается класс *FragmentStateAdapter*:

```
import androidx.annotation.NonNull;
import androidx.fragment.app.Fragment;
import androidx.fragment.app.FragmentActivity;
import androidx.viewpager2.adapter.FragmentStateAdapter;

public class ViewPageAdapter extends FragmentStateAdapter {
    private static final int CARD_ITEM_SIZE = 10;
    public ViewPageAdapter(@NonNull FragmentActivity fragmentActivity)
```

```
{
        super(fragmentActivity);
    }
    @NonNull @Override public Fragment createFragment(int position) {
        return CardFragment.newInstance(position);
    }
    @Override public int getItemCount() {
        return CARD_ITEM_SIZE;
    }
}
```

Затем создается класс активности *MainActivity:*

```
import android.os.Bundle;

import java.util.ArrayList;

public class MainActivity extends AppCompatActivity {

 ViewPager2 viewPager;
@Override
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        viewPager = findViewById(R.id.view_pager);
        viewPager.setAdapter(createCardAdapter());
        }
private ViewPageAdapter createCardAdapter() {
        ViewPageAdapter adapter = new ViewPageAdapter(this);
        return adapter;
        }
}
```

В *onCreate* создается и устанавливается адаптер для *ViewPager*. После этого следует запустить приложение, результаты его отображены на рис. 7.8.

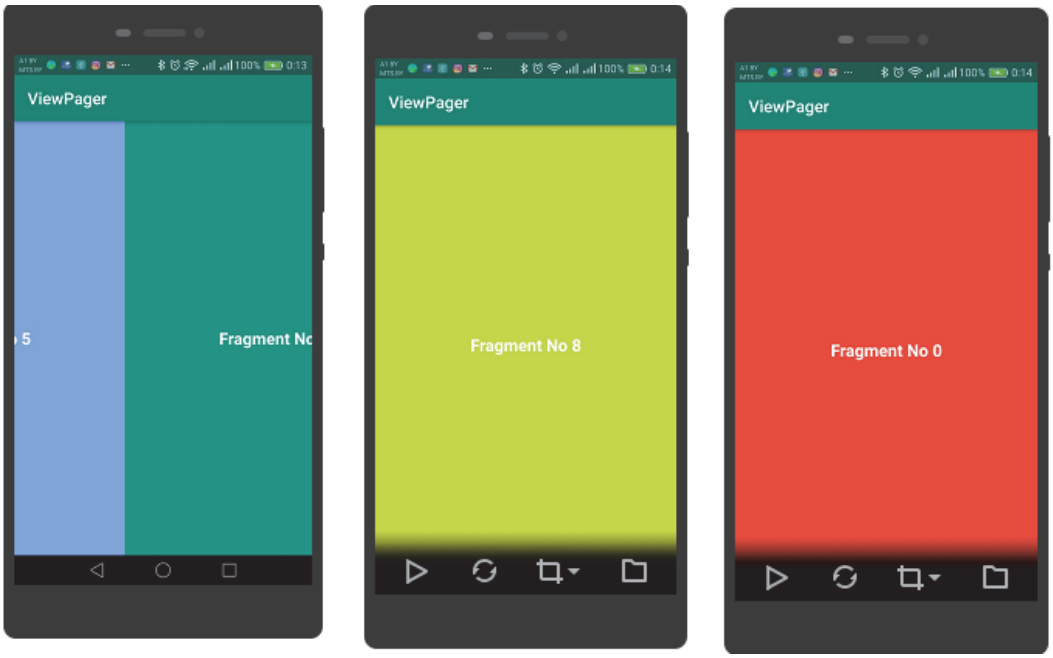


Рис. 7.8. Экраны приложения с *ViewPager*

8. РАБОТА С ФАЙЛОВОЙ СИСТЕМОЙ

Ядро Linux, используемое в Android, в той или иной степени поддерживает большое количество самых разных файловых систем от используемых в Windows FAT и NTFS, до сетевой 9pfs из Plan 9. Есть и много «родных» для Linux файловых систем – например, Btrfs и семейство ext. Стандартом де-факто для Linux уже долгое время является **ext4**, используемая по умолчанию большинством популярных дистрибутивов Linux. В некоторых сборках также используется **F2FS** (Flash-Friendly File System), оптимизированная специально для flash-памяти.

Приложения, их APK, файлы odex (скомпилированный ahead-of-time Java-код) и ELF-библиотеки устанавливаются в */system/app* (для приложений, поставляемых с системой) или в */data/app* (для установленных пользователем приложений). Каждому предустановленному приложению при создании сборки Android выделяется папка с именем вида */system/app/Terminal*, а для устанавливаемых пользователем приложений при установке создаются папки, имена которых начинаются с их имени пакета.

Содержимое /*system* описывает систему и содержит большинство составляющих ее файлов. Категория /*system* располагается в отдельном разделе flash-памяти, который по умолчанию монтируется в режиме read-only; обычно данные на нем изменяются только при обновлении системы. Категория /*data* также располагается в отдельном разделе и описывает изменяемое состояние конкретного устройства, в том числе пользовательские настройки, установленные приложения и их данные, кеши и т. п. Очистка всех пользовательских данных заключается просто в очистке содержимого раздела *data*; нетронутая система остается установлена в разделе *system*.

Для хранения изменяемых данных каждому приложению выделяется папка в */data/data*. Доступ к этой папке есть только у самого приложения – то есть только у UID, под которым запускается это приложение (если приложение использует несколько UID или несколько приложений используют общий UID). В этой папке приложения сохраняют настройки, кеш (в подпапках *shared_prefs* и *cache* соответственно) и любые другие нужные им данные.

Система знает о существовании папки *cache* и может очищать ее самостоятельно при нехватке места. При удалении приложения вся его папка полностью удаляется, и приложение не оставляет за собой следов. Альтернативно и то, и другое пользователь может явно сделать в настройках.

Все устройства Android имеют две области хранения файлов: *внутреннюю память* и *внешние хранилища*. Эти области появились в первые годы существования Android, когда на большинстве устройств имелись встроенная память (внутреннее хранилище) и карты памяти, например micro SD (внешнее хранилище). Некоторые устройства делят встроенную память на внутренний и внешний разделы, так что даже без съемных носителей в системе имеется две области хранения файлов, и API-интерфейс работает одинаково вне зависимости от типа внешнего хранилища.

Внешнее хранилище играет роль домашней папки пользователя – именно там располагаются такие папки, как *Documents*, *Download*, *Music* и *Pictures*.

В отличие от внутреннего хранилища, разделенного на папки отдельных приложений, внешнее хранилище представляет собой «общую зону». К нему есть полный доступ у любого приложения, получившего соответствующее разрешение от пользователя.

Исходно предполагалось, что внешнее хранилище действительно будет располагаться на внешней SD-карте, поскольку в то время объем SD-карт значительно превышал объем встраиваемой в телефоны памяти. С тех пор условия изменились; в современных телефонах часто нет слота для SD-карты, зато устанавливается огромное количество встроенной памяти.

Поэтому в современном Android практически всегда и внутреннее, и внешнее хранилища располагаются во встроенной памяти. Настоящий путь, по которому располагается внешнее хранилище в файловой системе, имеет форму /*data*/*media*/0 (для каждого пользователя устройства создается отдельное внешнее хранилище). В целях совместимости до внешнего хранилища также можно добраться по путям /*sdcard*, /*mnt*/*sdcard*, /*storage*/*self*/*primary*, /*storage*/*emulated*/0, нескольким путям, начинающимся с /*mnt*/*runtime*/, и др.

С другой стороны, у многих устройств все-таки есть слот для SD-карты. Вставленную в Android-устройство SD-карту можно использовать как обычный внешний диск. Кроме того, Android позволяет «заимствовать» SD-карту и разместить внутреннее и внешнее хранилище на ней (это называется заимствованным хранилищем – *adopted storage*). При этом система переформатирует SD-карту и шифрует содержимое – хранящиеся на ней данные невозможно прочесть, подключив ее к другому устройству.

8.1. Internal Storage (внутренняя память)

Внутренняя память имеет следующие характеристики:

— энергозависимая;

— хранятся apk-файлы, данные приложений, медиафайлы, документы и пр.;

— сохраненные данные в памяти позволяют читать и записывать файлы;

— файлы могут быть доступны только данному приложению, а не другим приложениям. Приложение всегда имеет разрешение на чтение и запись файлов в свой внутренний каталог на *Internal Storage*. Не требует разрешений для обращения;

— файлы хранятся до тех пор, пока приложение остается на устройстве. Когда оно удаляется, все связанные файлы автоматически удаляются (рис. 8.1);

— файлы хранятся в разделе *data*/*data*/, за которым следует имя пакета приложения;

— пользователь может явно разрешить другим приложениям доступ к файлам;

— чтобы сделать данные конфиденциальными, можно использовать режим *MODE_PRIVATE*;

— данные хранятся в файле в битовом формате, поэтому требуется преобразовать их перед добавлением в файл или извлечением из него;

— для непосредственного чтения и записи файлов применяются стандартные классы Java из пакета *java.io*.

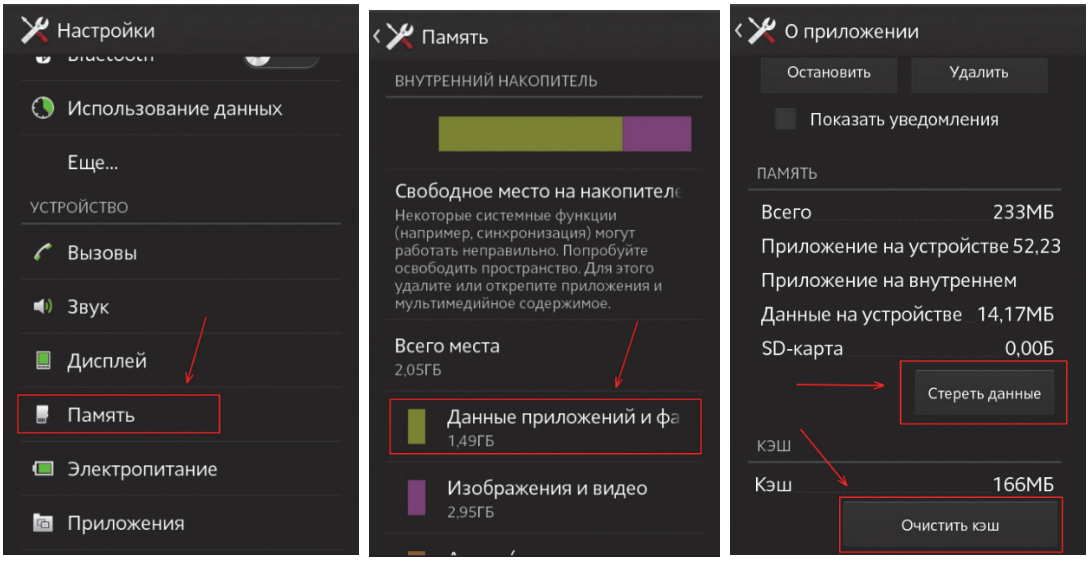


Рис. 8.1. Доступ к внутренней памяти устройства

Внутренняя память лучше всего подходит для ситуаций, когда надо быть уверенным, что ни пользователь, ни другие приложения не смогут получить доступ к файлам. Есть следующие режимы:

– *MODE_PRIVATE* – в этом режиме данные, сохраненные ранее, всегда переопределяются текущими данными, т. е. каждый раз, когда добавляется новая запись в файл, предыдущий контент удаляется или переопределяется (режим по умолчанию);

– *MODE_APPEND* – в этом режиме данные добавляются к существующему контенту.

Запись данных в файл во внутреннем хранилище может быть выполнена следующим образом:

```
String File_Name= "Demo.txt"; // Имя
String Data="Hello!!"; // Данные

  FileOutputStream fileobj = null;
try {
    fileobj = openFileOutput(File_Name, Context.MODE_PRIVATE);

    byte[] ByteArray = Data.getBytes(); //Конвертируем в bytes stream
    fileobj.write(ByteArray); //Записываем в файл
    fileobj.close(); // Закрываем файл
}
catch (FileNotFoundException e  ) {
     e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
```

Для того чтобы проверить записанные данные, справа в Android Studio есть вкладка *Device File Systsem*. На открывшейся панели можно найти записанный файл (рис. 8.2). Для этого следует перейти в *data/data/«имена_пакетов»/files/Demo.txt*. Также есть возможность открыть и посмотреть содержимое файла.

Для просмотра через эмулятор можно вызвать команду *Tools* → *Android* → *Android Device Monitor* (рис. 8.3). Затем выбрать процесс и открыть файл.

Ниже рассмотрены основные операции с файлами.

Функция создания файла:

```
public static void createFile(String filesDir,String fileName){
    try {
            File file = new File(filesDir, fileName);
            file.createNewFile();
            Log.d("Log_2", "Файл " + fileName + " создан");
    }catch (IOException e){
        Log.d("Log_2", "Файл " + fileName + " не создан");
    }
}
```

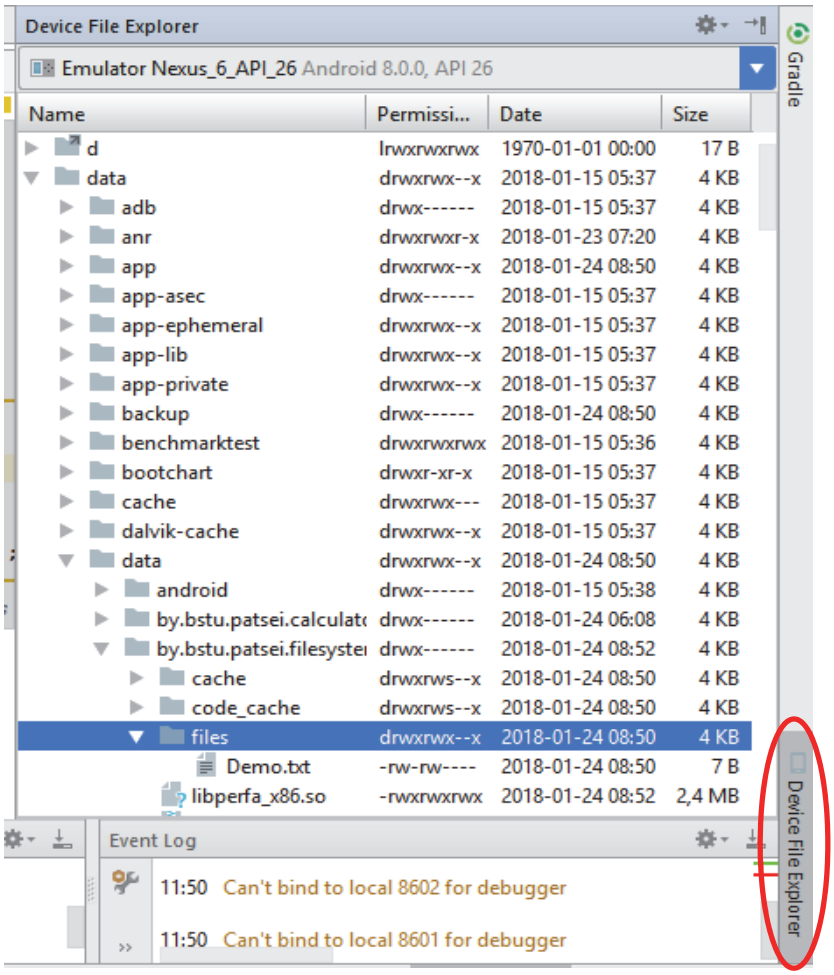


Рис. 8.2. Панель просмотра файлов устройства

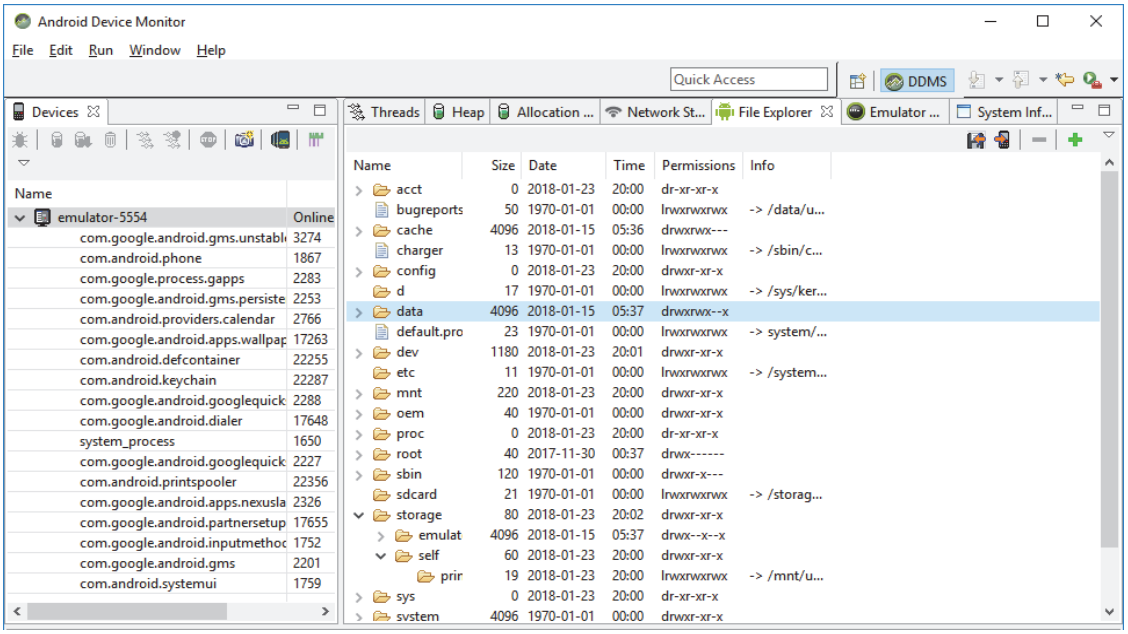


Рис. 8.3. Просмотр файлов через *Android Device Monitor*

Функция записи в файл:

```
public static void writeToFile(String filesDir, String fileName,String
string){
    if(!isExistFile(filesDir,fileName)){
       createFile(filesDir,fileName);
    }
    File file = new File(filesDir,fileName);
    BufferedWriter bw;
    try{
        FileWriter fw = new FileWriter(file,true);
        bw = new BufferedWriter(fw);
        bw.write(string);
        bw.close();
        Log.d("Log_2", "Файл " + fileName + " записан");
    }
    catch (IOException e){
        Log.d("Log_2", "Файл " + fileName + " не открыт " +e.getMes-
sage() );
    }
}
```

Функция чтения файла:

```
public static String readFromFile(Context context,String fileName){

    String line="";
    try {
        FileInputStream fis = context.openFileInput(fileName);
        InputStreamReader isr = new InputStreamReader(fis, "UTF-8");
        BufferedReader bufferedReader = new BufferedReader(isr);
        StringBuilder sb = new StringBuilder();
        while ((line = bufferedReader.readLine()) != null) {
            sb.append(line).append("\n");
        }
        line = sb.toString();
    } catch (FileNotFoundException e) {
    } catch (UnsupportedEncodingException e) {
    } catch (IOException e) {
    }
    return line;
}
```

Функция записи в файл произвольного доступа:

```
public static void writeToRandomAccessFile(String filesDir, String
fileName,String string, long point){

    if(!isExistFile(filesDir,fileName)){
        createFile(filesDir,fileName);
    }
    File file = new File(filesDir,fileName);
    try {
      RandomAccessFile randomAccessFile = new RandomAccessFile(file,
"rw"); // r - read, rw - read and write
        randomAccessFile.seek(point);
```

```
            randomAccessFile.writeChars(string);
            Log.d("Log_2", "Файл " + fileName + " записан");
        }
        catch (FileNotFoundException e){
            Log.d("Log_2", "Файл " + fileName + " не найден " +e.getMessage() );
        }
        catch (IOException e){
            Log.d("Log_2", "Файл " + fileName + " не открыт " +e.getMessage() );
        }

}
```

Функция чтения из файла произвольного доступа:

```
public static String readFromRandomAccessFile(String filesDir, String
fileName,long point, int length){

    File file = new File(filesDir, fileName);
    byte [] buffer = new byte[length];
    String data="";
try{
  RandomAccessFile randomAccessFile = new RandomAccessFile(file,"r");
        randomAccessFile.seek(point);
        randomAccessFile.read(buffer);
        data = new String(buffer,"utf-16");
 }
catch (FileNotFoundException e){
  Log.d("Log_2", "Файл " + fileName + " не найден " +e.getMessage() );
}
catch (IOException e){
  Log.d("Log_2", "Файл " + fileName + " не открыт " +e.getMessage());
}
return data;
}
```

Функция удаления файла:

```
public static void deleteFile(String filesDir,String fileName){
    File file = new File(filesDir,fileName);
    file.delete();
}
```

Если нужно временно сохранить некоторые данные, можно использовать специальный кеш-каталог. Когда пользователь удаляет приложение, эти файлы также удаляются.

8.2. External Storage (внешняя память)

Внешняя память имеет следующие характеристики:

— энергозавимая;

— может быть в собственной памяти устройства (эмуляция) или на внешнем носителе (SD-карте);

— доступна не всегда, пользователь может подключать и отключать хранилища, например USB-накопители;

— для доступа к *External*-памяти требуется разрешение, устанавливаемое в файле манифеста приложения;

— хранилища доступны для чтения везде, поэтому невозможно контролировать чтение сохраненных в них данных;

— при удалении пользователем приложения система Android удаляет из внешних хранилищ файлы этого приложения, только если они сохраняются в директории из *getExternalFilesDirectory*();

— доступность памяти должна проверяться;

— может содержать *public*- и *private*-файлы.

Внешнее хранилище лучше всего подходит для файлов без ограничений доступа и файлов, которые нужно сделать доступными другим приложениям или пользователю.

Для записи во внешнее хранилище следует указать запрос разрешения *WRITE_EXTERNAL_STORAGE* в файле манифеста:

```
<manifest ...>
    <uses-permission android:name="android.permission.WRITE_EXTER-
NAL_STORAGE" />
    ...
</manifest>
```

Если приложению потребуется считать данные из внешнего хранилища (но не записывать их), необходимо будет декларировать разрешение *READ_EXTERNAL_STORAGE*:

```
<manifest ...>
   <uses-permission android:name="android.permission.READ_EXTER-
NAL_STORAGE" />
    ...
</manifest>
```

Однако если приложение использует разрешение *WRITE_EXTERNAL_STORAGE*, оно косвенно получает разрешение на чтение данных из внешнего хранилища.

Чтобы избежать сбоя приложения, необходимо проверить, доступна ли SD-карта хранения для операций чтения и записи. Метод *getExternalStorageState*() используется для определения состояния носителя данных:

```
boolean isAvailable = false;
boolean isWritable = false;
boolean isReadable = false;
String state = Environment.getExternalStorageState();
```

```
if(Environment.MEDIA_MOUNTED.equals(state)){
 // Read-  и  write-операции доступны
    isAvailable = true;
    isWritable = true;
    isReadable = true;

} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)){
    // Read-операция доступна
    isAvailable = true;
    isWritable = false;
    isReadable = true;

} else {
    // SD-карта не смонтирована
    isAvailable = false;
    isWritable = false;
    isReadable = false; }
```

Хотя внешнее хранилище может быть изменено пользователем и другими приложениями, существует две категории файлов, которые в нем можно сохранять, это общедоступные (*public*) и личные (*private*) файлы.

8.2.1. Общедоступные файлы

Это файлы, которые должны быть доступны другим приложениям и пользователю. Когда пользователь удаляет приложение, эти файлы должны оставаться доступны пользователю. Например, в эту категорию входят снимки, сделанные с помощью вашего приложения, а также другие загружаемые файлы.

8.2.2. Личные файлы

Это файлы, которые принадлежат приложению. Они должны удаляться при удалении приложения. Хотя технически эти файлы доступны для других приложений, поскольку находятся во внешнем хранилище, они не имеют никакой ценности для пользователей вне приложения. К этой категории относятся дополнительные ресурсы, загруженные приложением, и временные мультимедийные файлы.

Существуют следующие методы для работы с фйлами:

— *getExternalStorageDirectory*() – старый способ доступа к памяти, в настоящее время не рекомендуется. Получает прямую ссылку на *root*-директорию внейшней памяти;

— *getExternalFilesDir(String type)* – рекомендованный способ создания *private*-файлов;

— *getExternalStoragePublicDirectory*() – рекомендованный способ получения доступа к *public*-файлам. При деинсталляции приложения файлы не удаляются;

— *getFreeSpace*() и *getTotalSpace*() – методы позволяют узнать текущее доступное пространство и общее пространство в хранилище. Эта информация также позволяет избежать переполнения объема хранилища сверх определенного уровня.

```
TextView showText;

public void getPublic(TextView view) {
    File folder = Environment.getExternalStoragePublicDirectory(Envi-
ronment.DIRECTORY_ALARMS); // Имя каталога
    File myFile = new File(folder, "PublicData.txt"); // Имя файла
    String text = getdata(myFile);
    if (text != null) {
        showText.setText(text);
    } else {
        showText.setText("No Data");
    }
}

public void getPrivate(TextView view) {
    File folder = getExternalFilesDir("FSAndroid"); // Имя каталога
    File myFile = new File(folder, "PrivateData.txt"); // Имя файла
    String text = getdata(myFile);
    if (text != null) {
        showText.setText(text);
    } else {
        showText.setText("No Data");
    }
}

private String getdata(File myfile) {
    FileInputStream fileInputStream = null;
    try {
        fileInputStream = new FileInputStream(myfile);
        int i = -1;
        StringBuffer buffer = new StringBuffer();
        while ((i = fileInputStream.read()) != -1) {
            buffer.append((char) i);
        }
        return buffer.toString();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (fileInputStream != null) {
            try {
                fileInputStream.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    return null;
}
```

8.3. SharedPreferences

SharedPreferences (общие настройки) – тип файла, который позволяет сохранять информацию в формате «ключ – значение» для примитивных типов данных: *boolean*, *float*, *int*, *long* и *string*. Часто используется для сохранения настроек приложения.

Пары «ключ – значение» записываются в файлы XML, которые сохраняются в течение сеансов, даже если приложение закрыто. Можно вручную указать имя файла для сохранения.

8.3.1. Получение доступа

Для получения доступа к настройкам в коде приложения используются три метода:

– *getPreferences*() – вызывается внутри активности, чтобы обратиться к определенному для нее предпочтению; можно вызвать метод без указания названия настроек. Доступ к возвращенному ассоциативному массиву настроек ограничен активностью, из которой он был вызван. Каждая активность поддерживает только один безымянный объект *SharedPreferences*;

– *getSharedPreferences*() – вызывается внутри активности, чтобы обратиться к предпочтению на уровне приложения;

– *getDefaultSharedPreferences*() – вызывается из объекта *PreferencesManager*, чтобы получить общедоступную настройку, предоставляемую Android.

Все эти методы возвращают экземпляр класса *SharedPreferences*, из которого можно получить соответствующую настройку с помощью следующих методов:

– *getBoolean(String key, boolean defValue)*;
– *getFloat(String key, float defValue)*;
– *getInt(String key, int defValue)*;
– *getLong(String key, long defValue)*;
– *getString(String key, String defValue)*.

Чтобы создать или изменить *SharedPreferences*, нужно вызвать метод *getSharedPreferences* в контексте приложения, передав имя общих настроек (имя файла). Например, следующий код обращается к файлу *SharedPreferences*, который идентифицируется строкой ресурса *R.string.preference_file_key* и открывает его с помощью *private mode*. Поэтому файл доступен только вашему приложению:

```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences(
        getString(R.string.preference_file_key),Context.MODE_PRIVATE);
```

При именовании файлов *SharedPreferences* необходимо использовать имя, уникально идентифицируемое для приложения. Простой способ сделать это – давать префикс имени файла с идентификатором приложения. Например:

```
by.belstu.myapp.PREFERENCE_FILE_KEY
```

В качестве альтернативы, если нужен только один файл для *Activity*, можно использовать метод *getPreferences*():

```
SharedPreferences sharedPref = getPreferences(Context.MODE_PRIVATE);
```

Файлы настроек хранятся в каталоге */data/data/имя_пакета/shared_prefs/имя_файла_настроек.xml*. Поэтому в отладочных целях, если нужно сбросить настройки в эмуляторе, необходимо, используя файловый менеджер, зайти в нужную папку, удалить файл настроек и перезапустить эмулятор. На устройстве можно удалить программу и установить ее заново, то же самое можно сделать и на эмуляторе, что бывает проще, чем удалять файл настроек вручную и перезапускать эмулятор.

Если открыть файл настроек текстовым редактором, то можно увидеть приблизительно следующее:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="0005">|000000|000000|000000|0</string>
<string name="0004">|000000|000000|000000|1</string>
<string name="0006">|000000|000000|000000|0</string>
</map>
```

В данном случае в настройках хранятся только строковые значения.

8.3.2. Сохранение значений параметров

Для удобства следует создать константу для имени файла *SharedPreferences*, например:

```
// Это будет именем файла настроек
public static final String APP_PREFERENCES = "mysettings";
```

Далее нужно создать параметры для сохранения в настройках. Удобнее их сделать константами:

```
public static final String APP_PREFERENCES_NAME = "Nickname"; // Имя
public static final String APP_PREFERENCES_AGE = "Age"; // Возраст
```

Когда названия параметров определены, можно сохранять любые их значения. Для этого создается переменная, представляющая экземпляр класса *SharedPreferences*:

```
SharedPreferences  mSettings = getSharedPreferences(APP_PREFERENCES,
Context.MODE_PRIVATE);
```

В указанный метод передается название файла (он будет создан автоматически) и стандартное разрешение, дающее доступ только компонентам приложения.

Чтобы внести изменения в настройки (редактировать), нужно использовать класс *SharedPreferences.Editor*. Получить объект *Editor* можно через вызов метода *edit* объекта *SharedPreferences*, который нужно изменить. После того как внесены все необходимые изменения, следует вызвать метод *commit*() или *apply*() объекта *Editor*, чтобы изменения вступили в силу. Метод *apply*() работает в асинхронном режиме, что является более предпочтительным вариантом. Метод *commit*() приходится использовать для старых версий и, кроме того, он возвращает значение *true* в случае успеха и *false* в случае ошибки.

Предположим, что имеется два текстовых поля, где пользователь должен ввести имя и возраст. Чтобы сохранить параметр, необходимо получить текст, который ввел пользователь. Получив нужный текст, следует сохранить его через метод *putString*() (есть также *putLong*(), *putBoolean*() и т. п.):

```
SharedPreferences.Editor editor = mSettings.edit();
editor.putString(APP_PREFERENCES_NAME, "Nikita");
editor.apply();
```

Как правило, параметры в методах активности *onPause*() или *onStop*() сохраняются в тех случаях, когда между запусками приложения данные должны сохраняться. Но могут быть и другие сценарии.

8.3.3. Чтение значений параметров

Для считывания данных при загрузке приложения обычно используют методы *onCreate*() или *onResume*(). Чтобы получить доступ к настройкам программы и проверить, есть ли среди них нужный параметр, необходимо найти ключ *Nickname* и загрузить его значение в текстовое поле.

```
if(mSettings.contains(APP_PREFERENCES_NAME)) {
    nicknameText.setText(mSettings.getString(APP_PREFERENCES_NAME, ""));
```

После проверки существования параметра *APP_PREFERENCES_NAME* и получения его значения через *getString*(), следует передать ключ и значение по умолчанию (используется в том случае, если для данного ключа пока не сохранено никакое значение). Остается только загрузить полученный результат в текстовое поле.

Можно получить ассоциативный массив со всеми ключами и значениями через метод *getAll*(). После этого можно проверить наличие конкретного ключа с помощью метода *contains*():

```
Map<String, ?> allPreferences = mSettings.getAll();
boolean containtsNickName = mSettings.contains(APP_PREFERENCES_NAME);
```

8.3.4. Очистка значений

Для очистки значений используйте методы *SharedPreferences.Editor.remove*(*String key*) и *SharedPreferences.Editor.clear*().

Начиная с API 11 у класса *SharedPreferences* появился новый метод *getStringSet*(), а у класса *SharedPreferences.Editor* родственный ему метод *putStringSet*(). Данные методы позволяют работать с наборами строк, что бывает удобно при большом количестве настроек, которые нужно сразу записать:

```
Set<String> names = new HashSet<String>();
names.add("Olga");
names.add("Nikita");
names.add("Andrei");
editor = mSettings.edit();
editor.putStringSet("strSetKey", names);
editor.apply();
```

или считать:

```
Set<String> ret = mSettings.getStringSet("strSetKey", new
HashSet<String>());
for(String r : ret) {
    Log.i("Share", "Имя: " + r);
}
```

8.3.5. Удаление файла

Как упоминалось ранее, файл настроек хранится в */data/data/имя_пакета/shared_prefs/имя_файла_настроек.xml*. Можно удалить его программно, например:

```
File file = new File("/data/data/.../shared_prefs/файл.xml");
file.delete();
```

Данные могут оставаться в памяти и временном файле **.bak*. Поэтому даже после удаления файла, он может заново воссоздаться. Файл автоматически удалится при удалении программы.

8.3.6. Сохранение состояния активности

Если необходимо сохранить информацию, которая принадлежит активности и не должна быть доступна другим компонентам (например, переменным экземпляра класса), можно вызвать метод *Activity.getPreferences*() без указания названия. Доступ к возвращенному ассоциативному массиву ограничен активностью, из которой он был вызван. Каждая активность поддерживает только один безымянный объект *SharedPreferences*:

```
SharedPreferences activityPreferences =
                          getPreferences(Activity.MODE_PRIVATE);
 // Извлеквем редактор, чтобы изменить Общие настройки.
 SharedPreferences.Editor editor = activityPreferences.edit();
// Записываем новые значения примитивных типов в объект Общих
настроек.
 editor.putString("currentTextValue", "new text");
 // Сохраняем изменения.
 editor.commit();
```

Система сама сгенерирует имя файла из имени пакета с добавлением слова *_preferences*.

```
SharedPreferences sharedPreferences =
              PreferenceManager.getDefaultSharedPreferences(this);
SharedPreferences.Editor edit = sharedPreferences.edit();
edit.putBoolean("isDone", false);
edit.commit();
```

9. ФРАГМЕНТЫ (FRAGMENTS)

Фрагменты впервые появились в Android версии 3.0 (API 11) главным образом для обеспечения большей динамичности и гибкости пользовательских интерфейсов на больших экранах, например у планшетов. Поскольку экраны планшетов гораздо больше, чем у смартфонов, они предоставляют больше возможностей для объединения и перестановки компонентов пользовательского интерфейса (рис. 9.1). Фрагменты позволяют делать это, избавляя разработчика от необходимости управлять сложными изменениями в иерархии представлений. Разбивая макет активности на фрагменты, разработчик получает возможность модифицировать внешний вид активности в ходе выполнения и сохранять эти изменения в стеке переходов назад, которым управляет активность.

Рис. 9.1. Размещение фрагментов в активности

Например, новостное приложение может использовать один фрагмент для показа списка статей слева, а другой – для отображения статьи справа. Оба фрагмента отображаются в одной *Activity* рядом друг с другом. Каждый из них имеет собственный набор методов обратного вызова жизненного цикла и управляет собственными событиями пользовательского ввода. Таким образом, вместо применения разных *Activity* для выбора статьи и ее чтения пользователь может сделать это в рамках одной *Activity*.

Фрагмент (класс *Fragment*) представляет поведение или часть пользовательского интерфейса в активности. Разработчик может объединить несколько фрагментов в одну *Activity* для построения многопанельного пользовательского интерфейса и повторного использования фрагмента в нескольких *Activity*. Фрагмент можно рассматривать как модульную часть *Activity*. Такая часть имеет свой жизненный цикл и самостоятельно обрабатывает события ввода. Кроме того, ее можно добавить или удалить непосредственно во время выполнения *Activity*. Это нечто вроде вложенной *Activity*, которую можно многократно использовать.

Фрагмент всегда должен быть встроен в *Activity*, и на его жизненный цикл напрямую влияет жизненный цикл *Activity*. Например, когда *Activity* приостановлена, в том же состоянии находятся и все фрагменты внутри нее, а когда активность уничтожается, уничтожаются и все фрагменты. Однако пока *Activity* выполняется, можно манипулировать каждым фрагментом независимо, например добавлять или удалять его. Когда разработчик выполняет такие транзакции с фрагментами, он может также добавить их в стек переходов назад. Стек переходов назад позволяет пользователю вернуть транзакцию с фрагментом (выполнить навигацию в обратном направлении), нажимая кнопку *Назад*.

9.1. Жизненный цикл фрагмента

Для создания фрагмента необходимо создать подкласс класса *Fragment* (или его существующего подкласса). Класс *Fragment* имеет код, во многом схожий с кодом *Activity*. Он содержит методы обратного вызова, аналогичные методам активности. На практике, если требуется преобразовать существующее приложение так, чтобы в нем использовались фрагменты, достаточно просто переместить код из методов обратного вызова активности в соответствующие методы обратного вызова фрагмента.

Фрагмент может находиться в одном из трех состояний (рис. 9.2):

1) ***возобновлен*** – фрагмент виден во время выполнения *Activity*;

2) ***приостановлен*** – фрагмент на переднем плане и выполняется, а в фокусе другая *Activity*. Но *Activity*, содержащая прежний фрагмент, видна;

3) ***остановлен*** – фрагмент не виден. Остановленный фрагмент по-прежнему активен (вся информация о состоянии и элементах сохранена в системе). Однако он больше не виден пользователю и будет уничтожен в случае уничтожения *Activity*.

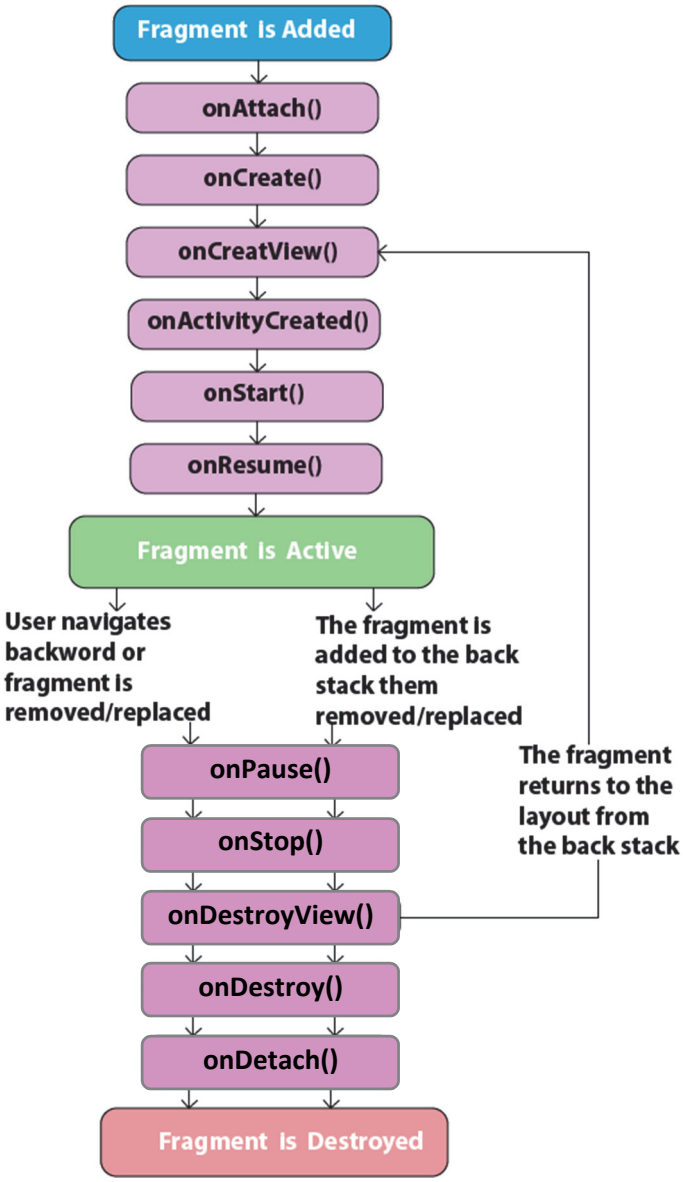


Рис. 9.2. Жизненный цикл фрагмента

Как правило, необходимо реализовать следующие методы жизненного цикла:

– *onCreate*() – система вызывает этот метод, когда создает фрагмент. В своей реализации разработчик должен инициализировать ключевые компоненты фрагмента, которые требуется сохранить, когда фрагмент находится в состоянии паузы или возобновлен после остановки;

– *onCreateView*() – система вызывает этот метод при первом отображении пользовательского интерфейса фрагмента на дисплее. Для прорисовки пользовательского интерфейса фрагмента следует возвратить из этого метода объект *View*, который является корневым в макете фрагмента. Если фрагмент не имеет пользовательского интерфейса, можно возвратить *null*;

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

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

– *onAttach*() – вызывается, когда фрагмент связывается с активностью (ему передается объект *Activity*);

– *onActivityCreated*() – вызывается, когда метод *onCreate*(), принадлежащий активности, возвращает управление;

– *onDestroyView*() – вызывается при удалении иерархии представлений, связанной с фрагментом;

– *onDetach*() – вызывается при разрыве связи фрагмента с активностью.

Зависимость жизненого цикла фрагмента от содержащей его активности иллюстрируется рисунком (рис. 9.3). На рисунке показано, что очередное состояние активности определяет, какие методы обратного вызова может принимать фрагмент. Например, когда активность принимает метод обратного вызова *onCreate*(), фрагмент внутри этой активности принимает метод обратного вызова *onActivityCreated*().

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

Однако если активность выходит из данного состояния, продвижение фрагмента по его жизненному циклу снова осуществляется активностью. Таким образом, *Activity* является хостом и управляет фрагментом.

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

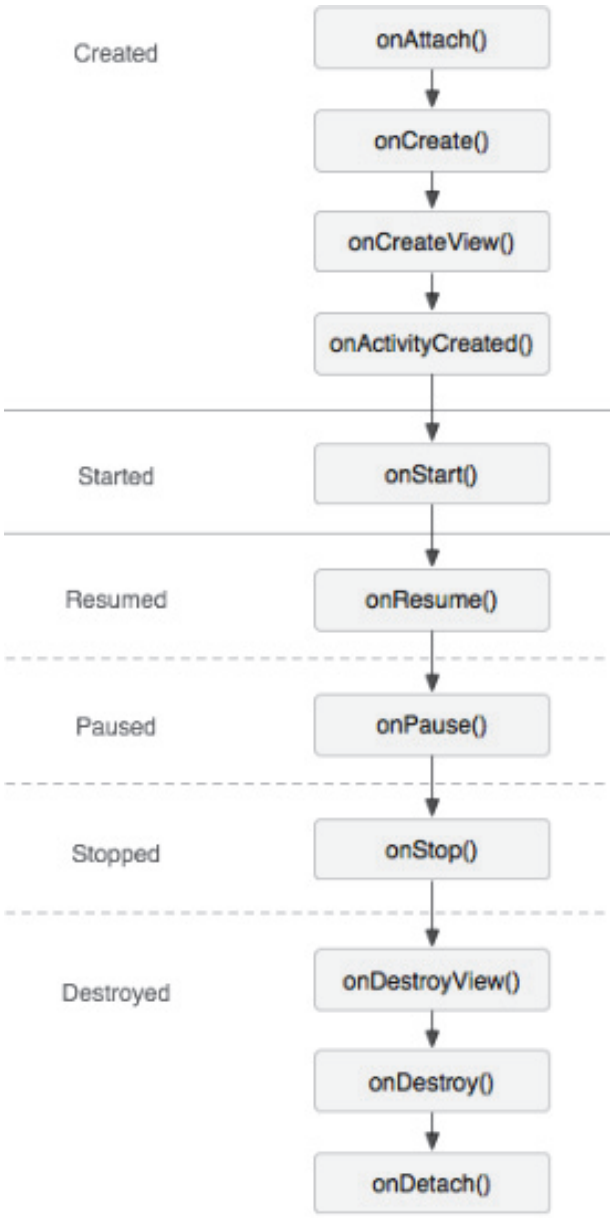


Рис. 9.3. Методы обратного вызова фрагмента

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

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

Сравнение жизненного цикла активности и фрагмента

Метод жизненного цикла	*Activity*	*Fragment*
onAttach()	–	+
onCreate()	+	+
onCreateView()	–	+
onActivityCreated()	–	+
onStart()	+	+
onPause()	+	+
onResume()	+	+
onStop()	+	+
onDestroyView()	–	+
onRestart()	+	–
onDestroy()	+	+
onDetach()	–	+

9.2. Разновидности и классы фрагментов

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

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

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

WebViewFragment – для взаимодействия между фрагментами используется класс *android.app.FragmentManager* (специальный менеджер по фрагментам).

Для транзакций (добавление, удаление, замена) используется класс-помощник *android.app.FragmentTransaction*. В 2018 г. Google объявила фрагменты из пакета *androd.app* устаревшими. Поэтому их необходимо заменять классами из библиотеки совместимости: *android.support.v4.app.FragmentActivity*; *android.support.v4.app.Fragment*; *android.support.v4.app.FragmentManager*; *android.support.v4.app.FragmentTransaction.*

9.3. Принципы работы с фрагментами

У каждого фрагмента должен быть свой класс, наследуемый от класса *Fragment* или схожих классов, о которых говорилось выше. Это похоже на создание новой активности или нового компонента.

Разметку для фрагмента можно создать программно или декларативно через XML.

Создание разметки для фрагмента ничем не отличается от создания разметки для активности.

9.3.1. Создание фрагмента

Можно рассмотреть создание фрагментов на примере. При запуске приложение открывает активность *MainActivity* (рис. 9.4). Активность использует два фрагмента *StudentListFragment* и *StudentDetailFragment*. Фрагмент *StudentListFragment* отображает список студентов. Фрагмент *StudentDetailFragment* отображает подробное описание студента. Оба фрагмента получают свои данные из *Student.java.*

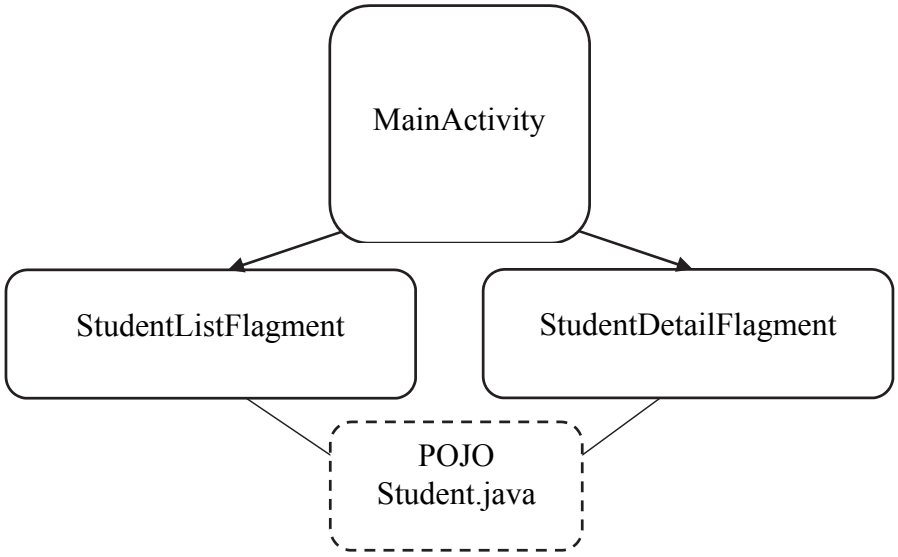


Рис. 9.4. Схема приложения с фрагментами

В приложении это будет выглядеть так, как изображено на рис. 9.5.

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

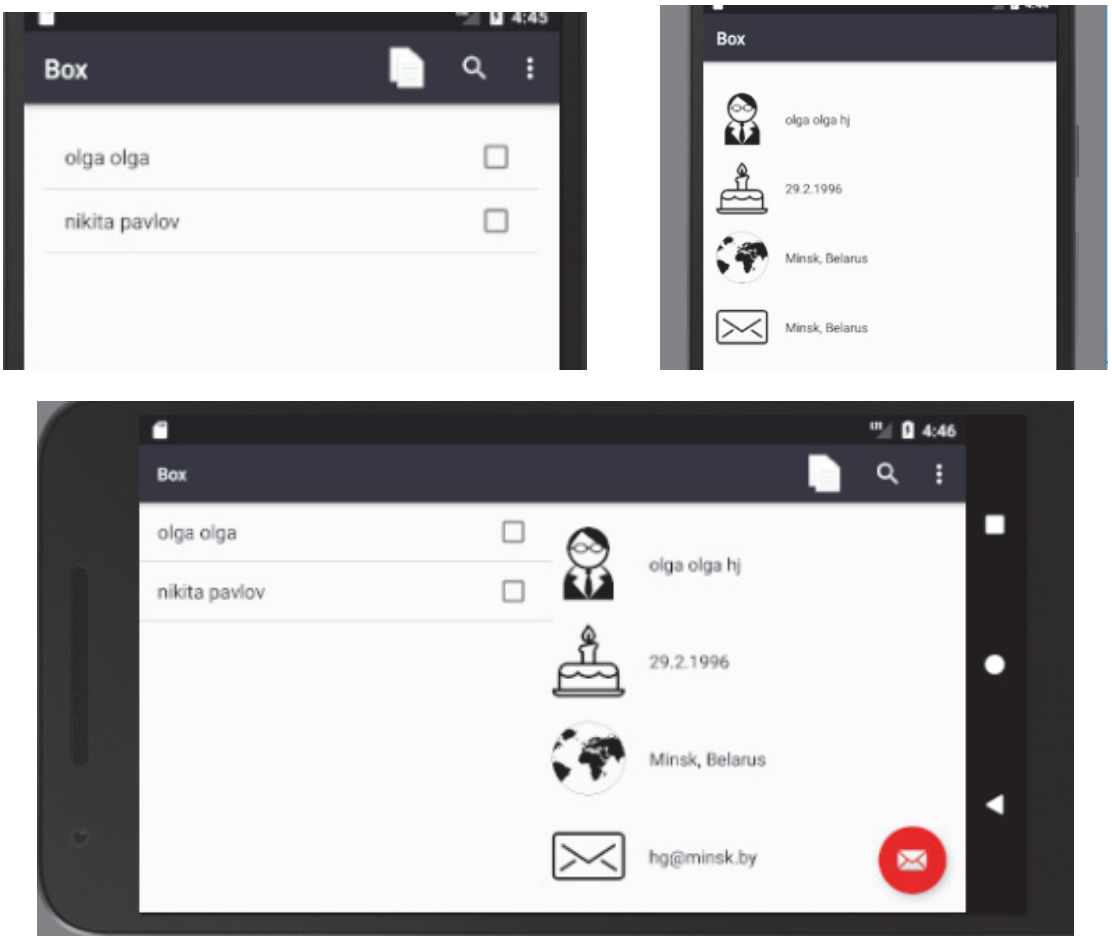


Рис. 9.5. Отображение приложения с фрагментами для разной ориентации

Для начала следует добавить класс *Student* в приложение. Это обычный файл модели, из которого приложение получает информацию о студентах. Класс определяет статический массив из трех элементов, как источник информации:

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

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

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

Теперь нужно добавить в проект новый фрагмент с именем *StudentDetailFragment*, предназначенный для вывода подробной информации о студенте. Новые фрагменты добавляются примерно так же, как и новые активности. В Android Studio следует выбрать команду *File* → *New...*→ *Fragment* → *Fragment* (*Blank*) и задать параметры нового фрагмента. Затем фрагменту присваивается имя, устанавливается флажок создания XML разметки и дается имя *fragment_student_detail* макету фрагмента (рис. 9.6).

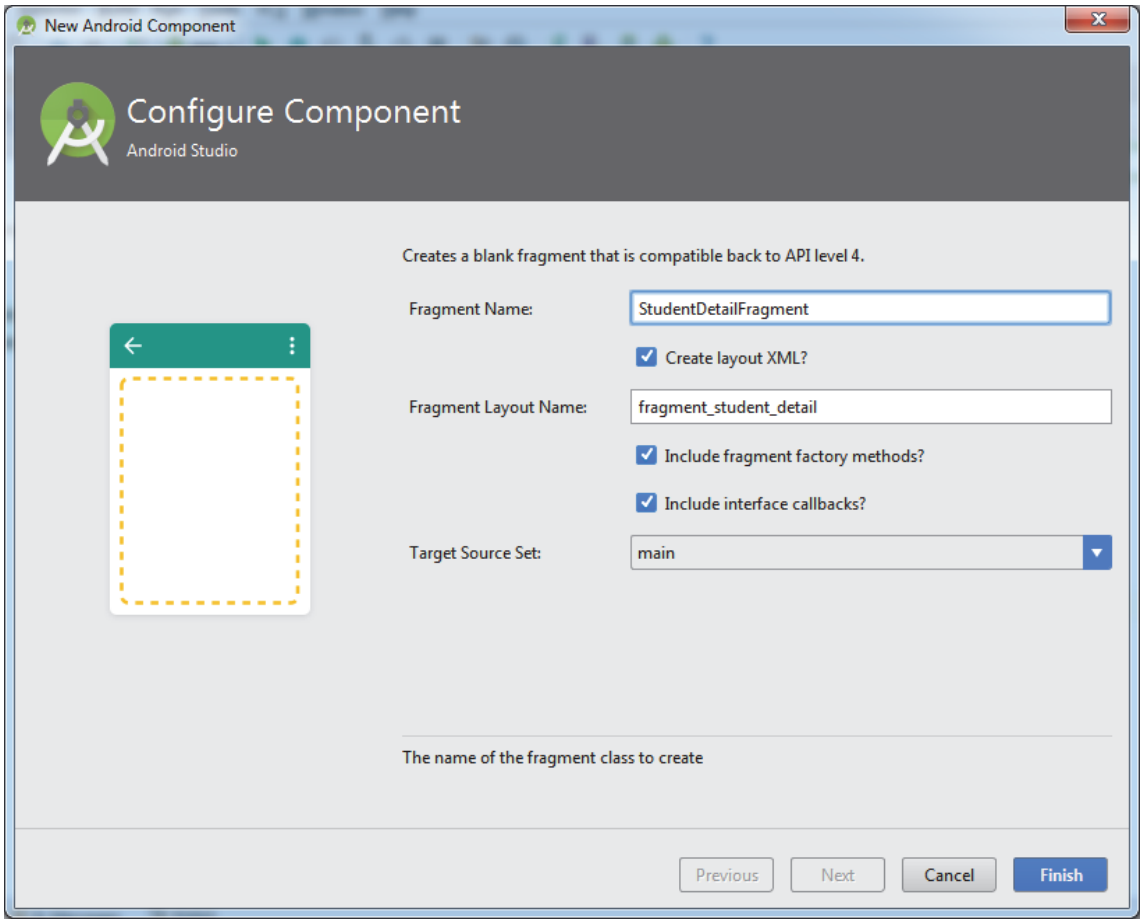


Рис. 9.6. Окно создания фрагмента

Таким образом, для создания фрагмента используется подкласс класса *Fragment*. Он содержит методы обратного вызова, которые были рассмотрены выше.

9.3.2. Макет фрагмента

Когда фрагмент добавлен как часть макета активности, он находится в объекте *ViewGroup* внутри иерархии представлений активности и определяет собственный макет. Разработчик может вставить фрагмент в макет активности двумя способами. Для этого следует объявить

фрагмент в файле макета активности как элемент <*fragment*> или добавить его в существующий объект *ViewGroup* в коде приложения. Можно также использовать фрагмент без интерфейса в качестве невидимого рабочего потока активности.

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

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

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

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

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

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

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

</LinearLayout>
```

Как показано выше, разметка макета фрагмента почти не отличается от разметки макета активности. Макет прост и состоит из четырех надписей.

При создании макетов фрагментов для своих приложений можно использовать все представления и макеты, которые уже использовались для построения активностей. Далее будет рассмотрен код самого фрагмента.

9.3.3. Код фрагмента

Код, сгенерированный Android Studio, нужно заменить следующим кодом.

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

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

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

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

9.3.4. Добавление фрагмента в макет активности

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

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

</RelativeLayout>
```

Макет состоит из единственного элемента <*fragment*>, который используется для добавления фрагмента в макет активности. Чтобы

указать, какой именно фрагмент должен использоваться, нужно присвоить атрибуту *class* полное имя фрагмента. В данном примере создается фрагмент *StudentDetailFragment*.

9.3.5. Обновление View-фрагмента

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

Для этого во фрагмент добавляется метод, который будет задавать значение идентификатора:

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

//...

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

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

9.3.6. Заполнение представлений в методе onStart() фрагмента

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

```
@Override
public void onStart() {

    super.onStart();

    View view = getView();

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

Фрагменты отличаются от активностей и поэтому не поддерживают некоторые методы. Например, у фрагментов отсутствует метод *find-ViewById*(). Чтобы получить ссылку на представление фрагмента,

сначала необходимо получить ссылку на корневое представление фрагмента методом *getView*() и по этой ссылке найти дочерние представления.

9.3.7. Взаимодействие активности и фрагмента

Для получения ссылки на фрагмент следует получить ссылку на ***диспетчера фрагментов*** активности при помощи метода *getSupportFragmentManager*(). Затем метод *findFragmentById*() используется для получения ссылки на фрагмент:

```
public class MainActivity extends AppCompatActivity {

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

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

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

Диспетчер фрагментов управляет всеми фрагментами, используемыми активностью.

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

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

9.3.8. Диспетчер фрагментов

Получить объект *FragmentManager* можно так:

```
FragmentManager fmsupp = getSupportFragmentManager();
FragmentManager fm = getFragmentManager();
```

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

– *findFragmentById*() (с UI) – находит фрагмент по идентификатору;

– *findFragmentByTag*() (без UI) – находит фрагмент по заданному тегу.

Для создания динамического UI используют разные методы управления транзакций:

– *add*() – добавляет фрагмент к активности;

– *remove*() – удаляет фрагмент из активности;

– *replace*() – заменяет один фрагмент на другой;

– *hide*() – прячет фрагмент (делает невидимым на экране);

– *show*() – выводит скрытый фрагмент на экран;

– *detach*() (API 13) – отсоединяет фрагмент от графического интерфейса, но экземпляр класса сохраняется;

– *attach*() (API 13) – присоединяет фрагмент, который был отсоединен методом *detach*().

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

Перед началом транзакции нужно получить эземпляр *FragmentTransaction* через метод *FragmentManager.beginTransaction*(). Далее вызываются различные методы для управления фрагментами. В конце любой транзакции, которая может состоять из цепочки вышеперечисленных методов, следует вызвать метод *commit*(). Например:

```
FragmentManager fragmentManager = getSupportFragmentManager();

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

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

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

9.3.9. Создание фрагмента со списком

Теперь необходимо создать второй фрагмент со списком студентов. После этого фрагменты можно будет использовать для создания разных пользовательских интерфейсов. Сначала создается фрагмент, содержащий списковое представление, и заполняется.

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

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

К тому же не надо реализовывать собственного слушателя событий. Класс *ListFragment* регистрируется как слушатель события для спискового представления и отслеживает щелчки. Чтобы фрагмент реагировал на щелчки, достаточно определить метод *onListItemClick*().

Списковые фрагменты добавляются в проект точно так же, как и обычные фрагменты. Для этого надо выбрать команду *File* → *New...* → *Fragment* → *Fragment* (*Blank*), присвоить фрагменту имя *StudentListFragment*, снять флажок создания XML-макета, а также флажок включения фабричных методов и интерфейсных обратных вызовов:

```
public class StudentListFragment extends ListFragment {

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

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

9.3.10. Использование ArrayAdapter для заполнения ListView

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

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

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

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

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

```
public class StudentListFragment extends ListFragment {

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup con-
tainer, Bundle savedInstanceState) {

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

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

Результат работы представлен на рис. 9.7.

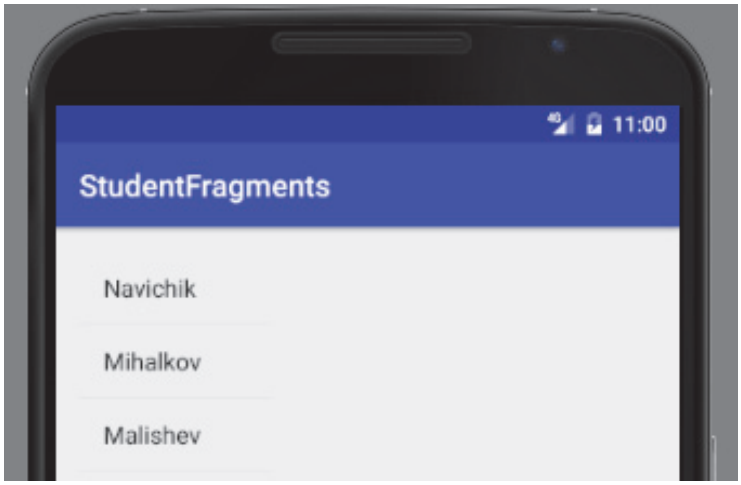


Рис. 9.7. Фрагмент со списком

9.3.11. Включение StudentListFragment в макет

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

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

Визуально получим следующее (рис. 9.8).

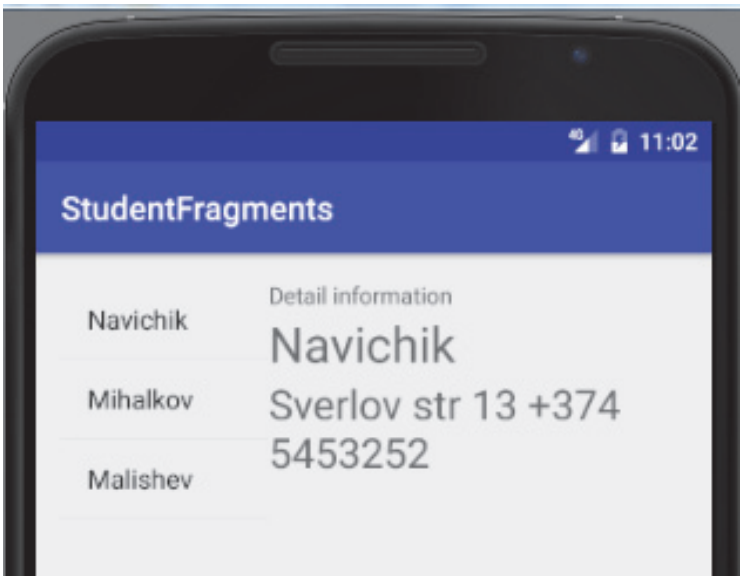


Рис. 9.8. Отображение двух фрагментов

9.3.12. Связывание списка с детализацией

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

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

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

Во время выполнения будет происходить следующая последовательность событий:

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

2) пользователь делает выбор в списке;

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

4) метод *onListItemClicked*() вызывает метод интерфейса с передачей идентификатора варианта.

На схеме жизненного цикла фрагмента видно, что присоединение его к активности вызывает метод *onAttach*() фрагмента с передачей объекта активности. Этот метод можно использовать для регистрации активности во фрагменте. Код выглядит следующим образом:

```
public class StudentListFragment extends ListFragment {

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

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

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

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

    @Override
    public void itemClicked(long id) {

    }

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

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

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

9.3.13. Поддержка кнопки возврата. Работа с Back stack

Предположим, пользователь выбирает одного студента, а потом другого. При нажатии кнопки *Назад* он рассчитывает вернуться к первому из выбранных им студентов (рис. 9.9).

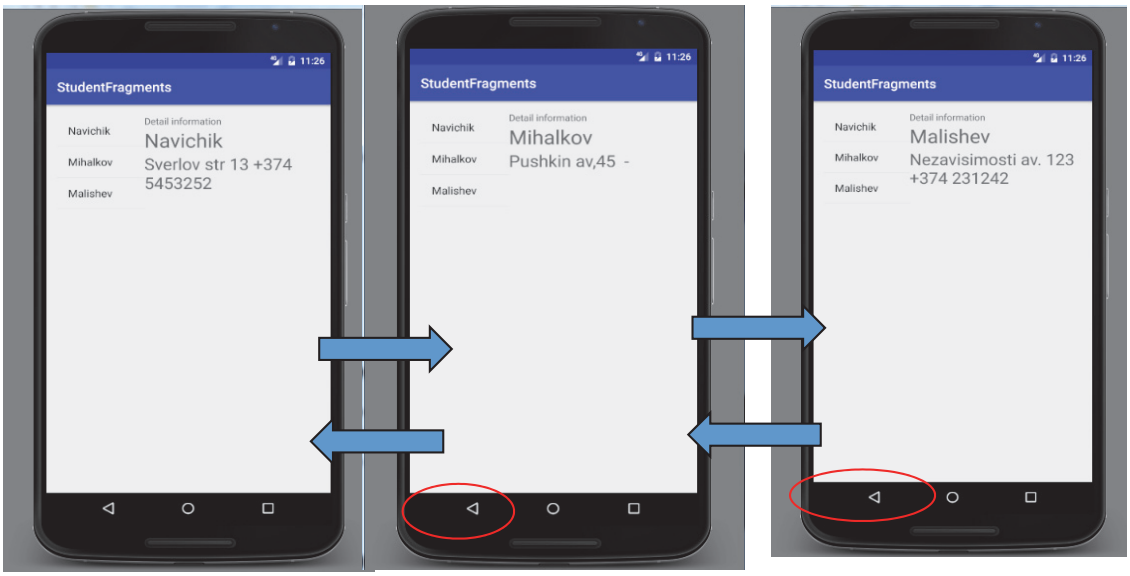


Рис. 9.9. Использование кнопки *Назад* для фрагментов

Стек возврата (Back stack) представляет собой список «мест», посещенных на устройстве. Каждое «место» представлено транзакцией

в стеке возврата. Многие транзакции осуществляют переход от одной активности к другой. Если нажать кнопку *Назад*, транзакция будет отменена, произойдет возврат к предыдущей активности. Однако транзакции в стеке возврата не обязаны быть переходами между активностями. Они могут быть простыми изменениями фрагментов на экране. Это означает, что смена фрагментов также может отменяться кнопкой *Назад*, как и смена активностей.

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

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

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

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

</LinearLayout>
```

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

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

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

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

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

Можно добавить фрагмент в контейнер или удалить:

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

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

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

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

Полный код реализации выглядит так:

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

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

```
        StudentDetailFragment details = new  StudentDetailFragment();

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

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

    }
```

Запустите приложение и проверьте кнопку возврата.

Если повернуть устройство, возникает проблема: какого бы студента ни выбрать, при повороте приложение всегда выводит информацию о нулевом студенте (рис. 9.10).

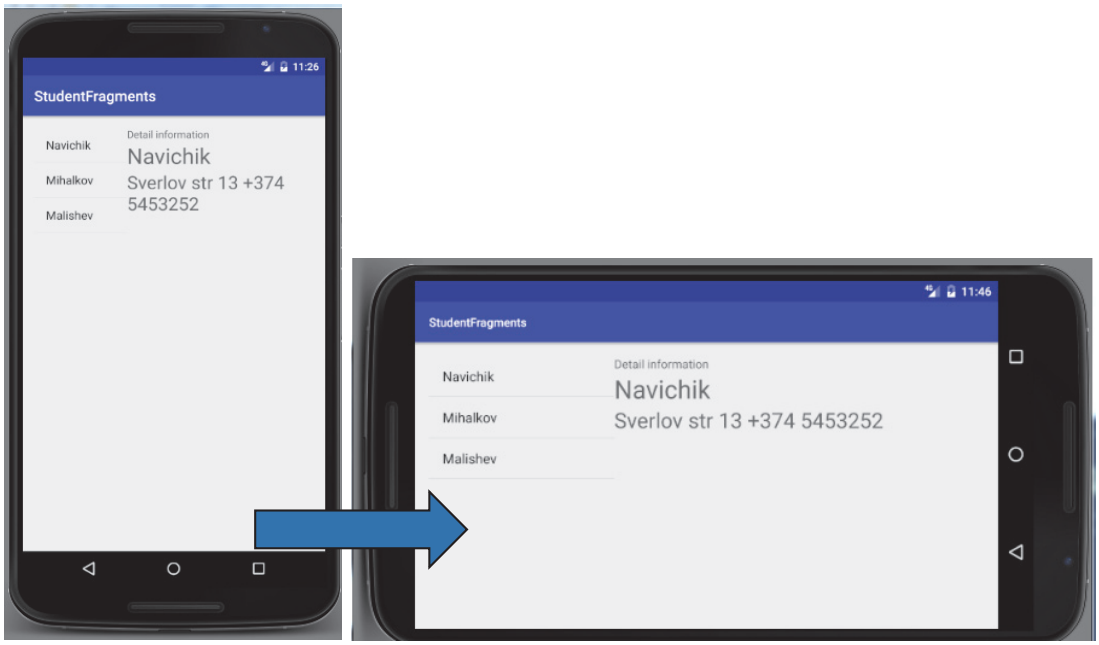


Рис. 9.10. Работа фрагметов при повороте утройства

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

Для фрагментов эта проблема решается примерно так же, как и для активностей. Сначала переопределяется метод *onSaveInstanceState*()

фрагмента и локальная переменная, значение которой требуется сохранить, помещается в параметр *Bundle* метода.

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

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

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

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup con-
tainer, Bundle savedInstanceState) {

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

9.3.14. Удержание состояния фрагментов

У фрагмента есть свойство *retainInstance*, которое по умолчанию содержит значение *false*. Это означает, что при поворотах фрагмент не сохраняется, а уничтожается и создается заново вместе с активностью-хостом. Вызов *setRetainInstance*(*true*) сохраняет фрагмент, который не уничтожается вместе с активностью, а передается новой активности в неизменном виде. Сделать это можно в методе *onCreate*(). Если значение *retainInstance* равно *true*, макет фрагмента уничтожается, но сам фрагмент остается. При создании новой активности новый экземпляр *FragmentManager* находит сохраненный фрагмент и воссоздает его макет.

Сохраненный фрагмент не уничтожается, а отсоединяется (detached) от «умирающей» активности. В сохраненном состоянии фрагмент все еще существует, но не имеет активности-хоста.

Переход в сохраненное состояние происходит только при выполнении двух условий:

1) для фрагмента был вызван метод *setRetainInstance*(*true*);

2) активность-хост уничтожается для изменения конфигурации (обычно поворот). Фрагмент находится в сохраненном состоянии очень недолго – с момента отсоединения от старой активности до повторного присоединения к новой, немедленно создаваемой активности.

Сохраненные фрагменты продолжают существовать только в случае уничтожения активности при изменении конфигурации. Если

активность уничтожается из-за того, что системе потребовалось освободить память, то все сохраненные фрагменты также будут уничтожены.

Реализованная схема с фрагментами представлена на рис. 9.4.

Однако чтобы выводить фрагменты в разных активностях, схема взаимодействия должна быть другая (рис. 9.11).

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

Чтобы создать один макет для устройств с большим экраном и еще несколько макетов для других устройств, необходимо макет для устройств с большим экраном поместить в папку *app/src/main/res/layout-large*, а макеты для других устройств – в папку *app/src/main/res/layout.*

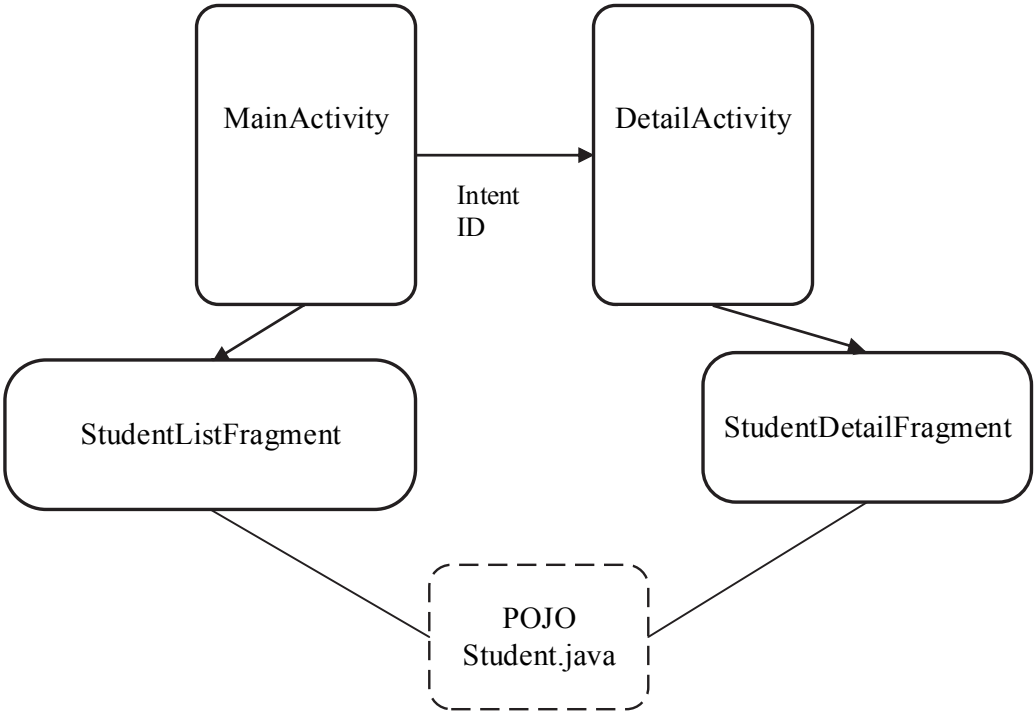


Рис. 9.11. Схема взаимодействия фрагментов в активности

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

Обе ситуации можно обработать в *MainActivity*; для этого следует проверить, какой макет используется устройством. Такая проверка может быть выполнена поиском представления с идентификатором *fragment_container*. Если *fragment_container* существует, значит,

устройство использует *activity_main.xml* из папки *layout-large*, следовательно, при выборе студента следует отобразить новый экземпляр *StudentDetailFragment*. Если же *fragment_container* не существует, то устройство использует версию *activity_main.xml* из папки *layout*, поэтому вместо этого запускается *DetailActivity*. Ниже приведен полный код *MainActivity.java*

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

 @Override
 public void itemClicked(long id) {

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

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

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

9.3.15. Аргументы фрагментов

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

— активность может создать фрагмент и установить аргументы для него;

— активность может вызвать методы экземпляра фрагмента;

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

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

Создать аргументы можно следующим образом:

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

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

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

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


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

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

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

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

В результате получаем вызов *onResume*() от ОС с хостом *Activity:*

```
@Override
public void onResume() {

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

    }
```

Фрагменты могут не только содержаться в активностях, но и вкладываться в другие фрагменты.

10. РАБОТА С БАЗОЙ ДАННЫХ SQLITE. АДАПТЕРЫ. ASYNCTASK

Механизм работы с базами данных в Android позволяет хранить и обрабатывать структурированную информацию. Любое приложение может создавать свои собственные базы данных, над которыми оно будет иметь полный контроль.

База данных SQLite доступна на любом Android-устройстве, ее не нужно устанавливать. Для работы большинства СУБД необходим специальный процесс сервера базы данных. SQLite обходится без сервера и представляет собой обычный файл.

Android хранит базы данных в каталоге */data/data/«имя_пакета»/databases* на эмуляторе (рис. 10.1), на устройстве путь может отличаться. Метод *Environment.getDataDirectory*() возвращает путь к каталогу *DATA*.

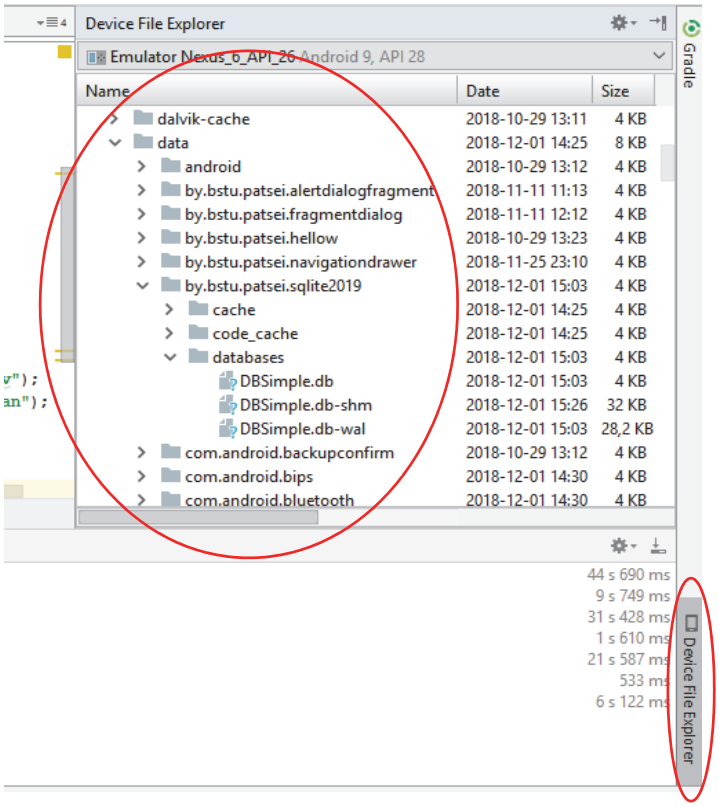


Рис. 10.1. Окно *Device File Explorer*

По умолчанию все базы данных закрытые, доступ к ним могут получить только те приложения, которые их создали.

Каждая база данных состоит из двух файлов. Имя первого файла базы данных соответствует имени базы данных. Это основной файл, в нем хранятся все данные. Второй файл – файл журнала. Его имя состоит из имени базы данных и суффикса *-journal*. В нем хранится информация обо всех внесенных изменениях. Если при работе с данными возникнет проблема, Android использует информацию журнала для отмены (или отката) последних изменений. Для просмотра можно воспользоваться *Device File Explorer* (см. рис. 10.1).

10.1. Программное обеспечение для работы с базами данных

В Android SDK включена оболочка *sqlite3.exe* для работы с командной строкой. Чтобы увидеть все возможности работы с оболочкой, необходимо зайти в *Android/android-sdk/platform-tools/*, запустить *sqlite3.exe* (рис. 10.2) и вызвать команду *help*.

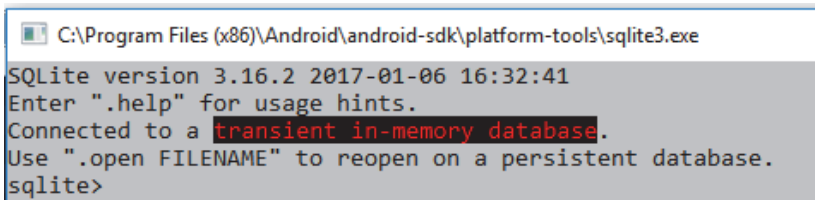


Рис. 10.2. Запуск *sqlite3.exe*

Существуют дополнительные инструменты: DbBrowser for SQLite, SQLite Expert Professional, SQLite Studio.

10.2. Определение схемы и контракта

В последних рекомендациях Google для работы с базой данных рекомендуется создать класс-контракт. Необходимо придерживаться этого правила.

Одним из основных принципов рекомендаций является схема: формальное объявление о том, как организована база данных. Схема отражена в операторах SQL, которые используются для создания базы данных.

Класс-контракт – это контейнер для констант, который определяет имена для *URI*, *таблиц* и *столбцов*. Класс контракта позволяет использовать одни и те же константы для всех других классов в одном пакете. Хороший способ организовать класс-контракт – поставить определения, которые являются глобальными для всей базы данных на корневом уровне класса. Затем для каждой таблицы создать внутренний класс, который перечисляет столбцы соответствующей таблицы.

В примере ниже определяется имя таблицы и имена столбцов для одной из них. В классе используется реализация интерфейса *BaseColumn*:

```
public final class DBContract {

        public DBContract() {}

        /* Внутренний класс определяет контент таблицы*/
   public static abstract class DBEntry implements BaseColumns {
          public static final String TABLE_NAME = "student";
          public static final String COLUMN_NAME_NAME = "name";
          public static final String COLUMN_NAME_INFO = "info";
          public   static   final  String  COLUMN_NAME_RATE  =  "rate";
        }
}
```

В большинстве случаев работа с базой данных происходит через специальные объекты *Cursor*, которые требуют наличия в таблице колонки с именем *_id*. Можно создать столбец вручную в коде, а можно положиться на *BaseColumn*, который создаст столбец с нужным именем автоматически. Если не работать с курсорами, то можно использовать и стандартное наименование *id* или вообще не пользоваться данным столбцом. Но это не желательно.

10.3. Классы для работы с SQLite

Система Android включает набор классов для управления базой данных. Основная часть этой работы выполняется тремя типами объектов.

Класс *SQLiteDatabase* можно сравнить с классом *SQLConnection* в спецификации JDBC. Он позволяет выполнять запросы к базе данных и различные манипуляции с ней. Ниже приведены методы:

– *query*() – чтение данных;

– *insert*() – вставка записей;

– *delete*() – удаление записей;
– *update*() – обновление записей;
– *execSQL*() – выполнение допустимого кода SQL;
– *rawQuery*() – выполнение «сырого» запроса.

Класс *SQLiteCursor* предназначен для чтения и записи данных. Его можно сравнить с классом *ResultSet* в JDBC. Вместо того чтобы извлекать данные и возвращать копию значений, курсоры ссылаются на результирующий набор исходных данных.

Класс *SQLiteQueryBuilder* используется для выпонения SQL-запросов. Сами SQL-выражения представлены классом *SQLiteStatement*, которые позволяют с помощью плейсхолдеров вставлять в выражения динамические данные.

Помощник SQLite создается расширением класса *SQLiteOpenHelper*. Он предоставляет средства для создания и управления базами данных и содержит два обязательных абстрактных метода:
– *onCreate*() – вызывается при первом создании базы данных;
– *onUpgrade*() – вызывается при модификации базы данных.

Также используются другие методы класса:
– *onDowngrade*(*SQLiteDatabase*, *int*, *int*);
– *onOpen*(*SQLiteDatabase*);
– *getReadableDatabase*();
– *getWritableDatabase*().

Для обслуживания базы данных создаются выражения создания и удаления:

```
private static final String SQL_CREATE_ENTRIES =
                "CREATE TABLE " + DBEntry.TABLE_NAME + " (" +
                 DBEntry._ID + " INTEGER PRIMARY KEY," +
                 DBEntry.COLUMN_NAME_NAME +"TEXT" + "," +
                 DBEntry.COLUMN_NAME_INFO +"TEXT" + "," +
                 DBEntry.COLUMN_NAME_RATE + "INTEGER)";
private static final String SQL_DELETE_ENTRIES =
               "DROP TABLE IF EXISTS " + DBEntry.TABLE_NAME;
```

И собственно класс самого помошника:

```
public class DbHelper  extends SQLiteOpenHelper {

        public static final int DATABASE_VERSION = 2;
        public static final String DATABASE_NAME = "DBSimple.db";
        public DbHelper(Context context) {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
        }
```

```
        public void onCreate(SQLiteDatabase db) {
            db.execSQL(SQL_CREATE_ENTRIES);
        }

public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
{
            db.execSQL(SQL_DELETE_ENTRIES);
            onCreate(db);
        }
    }
```

Константа *DATABASE_NAME* отвечает за имя файла, в котором будет храниться база данных приложения. Можно придумать любое имя и обойтись без расширения.

Вторая константа *DATABASE_VERSION* отвечает за номер версии базы и начинается с 1. Как только программа заметила обновление номера базы, она запускает метод *onUpgrade*(), который сформировался автоматически. В этом методе размещен код, срабатывающий при обновлении базы.

Если номер базы данных меньше, то вызывается метод *onDowngrade*() (но так происходит редко).

Метод *onCreate*() вызывается, если в устройстве нет базы данных и класс должен создать ее. У метода есть параметр *db*, относящийся к классу *SQLiteDatabase*. У класса есть специальный метод *execSQL*(), которому нужно передать запрос (SQL-скрипт) для создания таблицы.

Чтобы использовать реализацию вспомогательного класса, необходимо создать новый экземпляр, передать его конструктору контекст, можно имя базы данных, можно текущую версию, вызвать метод *getReadableDatabase*() или *getWritableDatabase*(), чтобы открыть и вернуть экземпляр базы данных. Вызов метода *getWritableDatabase*() может завершиться неудачно из-за проблем с полномочиями или нехваткой места на диске, поэтому лучше предусмотреть откат к методу *getReadableDatabase*(). Методы имеют разные названия, но возвращают один и тот же объект.

Метод для чтения *getReadableDatabase*() сначала проверит доступность на чтение/запись. В случае ошибки он проверит доступность на чтение и только потом уже вызовет исключение.

Второй метод сразу проверяет доступ к чтению/записи и вызывает исключение при отказе в доступе.

Рекомендуется создавать для каждой таблицы отдельный класс.

10.4. Режимы работы с базой данных

Для размещения информации в базе данных (вставки) создаем экземпляр:

```
DbHelper mDbHelper = new DbHelper(this);
```

При добавлении данных в таблицы применяются объекты класса *ContentValues*. Каждый такой объект содержит данные одной строки в таблице и, по сути, является ассоциативным массивом с именами столбцов и соответствующими значениями. Поэтому для вставки сначала подготавливаются данные с помощью класса *ContentValues*. Указывается имя колонки таблицы и значение по принципу ключ – значение. Затем вызывается метод *insert*(), который помещает подготовленные данные в таблицу.

У метода *insert*() три аргумента. В первом указывается имя таблицы, в которую будут добавляться записи. В третьем указывается объект *ContentValues*. Второй аргумент используется для указания колонки. SQL не позволяет вставлять пустую запись, и если будет использоваться пустой *ContentValue*, то он указывается во втором аргументе *null* во избежание ошибки:

```
// Получаем репозиторий в режиме записи
        SQLiteDatabase db = mDbHelper.getWritableDatabase();

// Создаем новую запись
        ContentValues values = new ContentValues();
        values.put(DBContract.DBEntry.COLUMN_NAME_NAME, "Ivanov");
        values.put(DBContract.DBEntry.COLUMN_NAME_INFO, "BSTU");
        values.put(DBContract.DBEntry.COLUMN_NAME_RATE, "8");

// Вставляем данные, и возвращается  primary key
        long newRowId;
        newRowId = db.insert(
                DBContract.DBEntry.TABLE_NAME,
                null,
                values);
```

Метод *insert*() возвращает идентификатор *_id* вставленной строки или –1 в случае ошибки.

Существует также второй способ вставки через метод *execSQL*(), когда подготавливается нужная строка и запускается скрипт. В этом варианте используется традиционный SQL-запрос *INSERT INTO...*

Изменение данных. Если запись уже существует, но нужно изменить какое-то значение, то вместо *insert*() используется метод *update*(). В остальном принцип тот же:

```
SQLiteDatabase db = mDbHelper.getWritableDatabase();

// Новое значение
    String title = "Murkov";
    ContentValues values = new ContentValues();
    values.put(DBEntry.COLUMN_NAME_NAME, title);

// Столбец, который надо обновлять
    String selection = DBEntry.COLUMN_NAME_NAME + " LIKE ?";
    String[] selectionArgs = { "Ivanov" };

    int count = db.update(
            DbHelper.DBEntry.TABLE_NAME,
            values,
            selection,
            selectionArgs);
```

Чтение данных. Считывать данные также можно двумя способами. Результат возвращается в виде объекта *Cursor*. Методы объекта *Cursor* предоставляют различные возможности навигации: *moveToFirst*, *moveToNext*, *moveToPrevious*, *getCount*, *getColumnIndexOrThrow*, *getColumnName*, *getColumnNames*, *moveToPosition*, *getPosition*.

Первый способ. Для чтения используется метод *query*() с передачей критериев выделения желаемых столбцов. Метод сочетает элементы *insert*() и *update*():

```
// Определяем названия столбцов,
// которые нужны для  выполнения запроса
String[] projection = {
        BaseColumns._ID,
        DBEntry.COLUMN_NAME_NAME,
        DBEntry.COLUMN_NAME_INFO,
    ...
};

 String selection = DBEntry.COLUMN_NAME_NAME + " = ?";
         String[] selectionArgs = { "Ivanov" };

// Возвращается Cursor
         String sortOrder =
         DBEntry.COLUMN_NAME_INFO + " DESC";

         Cursor cursor = db.query(
         DBEntry.TABLE_NAME,                  // Имя таблицы
         projection,                          // Столбцы
         selection,                           // Столбцы для WHERE
         selectionArgs,                       // Значения для WHERE
         null,                                // Не группировать строки
         null,                                // Не фильтровать
         sortOrder                            // Порядок сортировки
         );
```

В метод *query*() передают семь параметров. Если какой-то параметр для запроса не нужен, то ставится *null*:

— *table* – имя таблицы, к которой передается запрос;

— *String*[] *columnNames* – список имен возвращаемых полей (массив). При значении *null* возвращаются все столбцы;

— *String whereClause* – параметр, формирующий выражение WHERE (исключая сам оператор WHERE). Значение *null* возвращает все строки. Например: *_id = 19 and summary = ?*;

— *String*[] *selectionArgs* – значения аргументов фильтра. Можно включить знак «?» в *whereClause*. Он подставляется в запрос из заданного массива;

— *String*[] *groupBy* – фильтр для группировки, формирующий выражение GROUP BY (исключая сам оператор GROUP BY). Если GROUP BY не нужен, передается null;

— *String*[] *having* – фильтр для группировки, формирующий выражение HAVING (исключая сам оператор HAVING). Если не нужен, передается *null*;

— *String*[] *orderBy* – параметр, формирующий выражение ORDER BY (исключая сам оператор ORDER BY). При сортировке по умолчанию передается *null*.

Чтобы получить все записи из нужных столбцов без условий, достаточно указать имя таблицы в первом параметре и строчный массив во втором. В остальных параметрах *null*.

Для просмотра строки в месте курсора используется один из методов перемещения *Cursor*, которые всегда нужно вызывать перед считыванием значений. Обычно следует начинать с вызова *moveToFirst*(), который помещает «позицию чтения» на первую запись в результатах. Для каждой строки значение столбца можно прочитать путем вызова одного из методов *Cursor*, например *getString*() или *getLong*(). Для каждого из методов *get* надо передать указатель желаемого столбца, который может вызвать *getColumnIndex*() или *getColumnIndexOrThrow*(). После завершения работы с курсором закрываются все связанные объекты:

```
List itemIds = new ArrayList<>();
while(cursor.moveToNext()) {
        long itemId = cursor.getLong(
                cursor.getColumnIndexOrThrow(DBEntry._ID));
        itemIds.add(itemId);
    }
cursor.close();
```

Для перемещения между записями в курсорах используются четыре основных метода: *moveToFirst*(), *moveToLast*(), *moveToPrevious*() и *moveToNext*().

Для чтения данных из текущей записи курсора используются методы *get**(). Точное название метода зависит от типа значения, которое следует прочитать: *getString*(), *getInt*() и т. д. Каждый из методов получает один параметр – индекс столбца:

```
String name = cursor.getString(0);
```

Второй способ использует сырой (raw) SQL-запрос. Сначала формируется строка запроса и отдается методу *rawQuery*().

Удаление данных. Метод *delete*() класса *SQLiteDatabase* работает по тому же принципу, как и метод *update*() – возвращает количество удаленных столбцов.

```
// Определение 'where' части запроса
String selection = DBEntry.COLUMN_NAME_NAME + " LIKE ?";
// Определение аргументов в  placeholder
    String[] selectionArgs = { "Ivanov" };
// SQL statement
    int deletedRows = db.delete(DBEntry.TABLE_NAME, selection,
                                                selectionArgs);
```

Можно открывать и создавать базы данных без помощи класса *SQLiteOpenHelper*, используя метод *openOrCreateDatabase*(), принадлежащий объекту *Context* приложения. Следует получить доступ к базе данных. Для начала нужно вызвать метод *openOrCreateDatabase*(), чтобы создать новую базу данных. Затем из полученного экземпляра базы данных следует вызвать *execSQL*(), чтобы выполнять команды на языке SQL.

```
private static final String DATABASE_NAME = "myDatabase.db";
    private static final String DATABASE_TABLE = "mainTable";
    private static final String DATABASE_CREATE = "create table " +
DATABASE_TABLE + " ( _id integer primary key autoincrement," +
                                "column_one text not null);";

    SQLiteDatabase myDatabase;

    private void createDatabase() {
        myDatabase = openOrCreateDatabase(DATABASE_NAME,
                                    Context.MODE_PRIVATE,  null);
        myDatabase.execSQL(DATABASE_CREATE);
    }
```

10.5. Представление прочитанных данных

Одно из возможных решений по представлению данных выглядит так: прочитать список (студентов) из базы данных и сохранить информацию в массиве, передаваемом адаптеру массива (рис. 10.3).

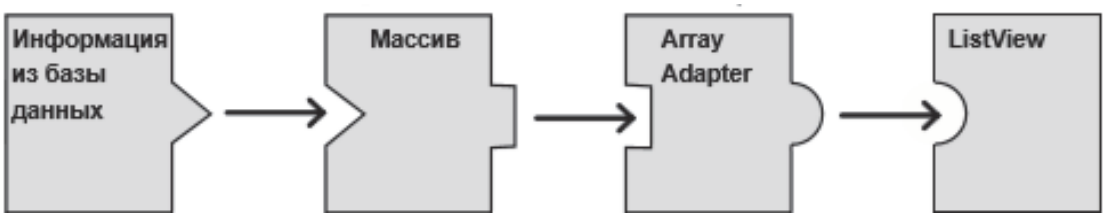


Рис. 10.3. Схема представления прочитанных данных на основе массива

При небольшой базе данных чтение всей информации и хранение ее в массиве (то есть в памяти) не создает проблем. Но если приложение работает с очень большим объемом информации, чтение из базы данных займет некоторое время. Кроме того, для хранения массива потребуется много памяти. Поэтому вместо класса адаптера массива *ArrayAdapter* используют адаптер курсора *CursorAdapter* (рис. 10.4).

Адаптер курсора читает столько данных, сколько нужно. В представлении *ListView* одновременно может отображаться лишь небольшая часть записей. На устройствах с маленьким экраном изначально могут выводиться, скажем, первые 10. Если бы данные хранились в массиве, то пришлось бы загрузить все записи из базы данных в массив и только потом вывести часть данных. С *CursorAdapter* все работает немного иначе.

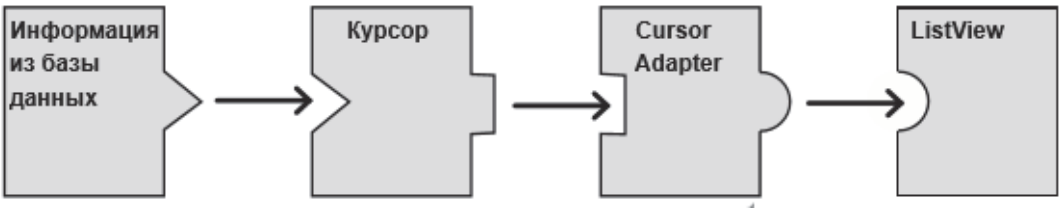


Рис. 10.4. Схема представления прочитанных данных на основе курсора

Когда списковое представление отображается впервые, оно масштабируется по размерам экрана. Предположим, в нем достаточно места для отображения пяти элементов. Компонент *ListView* не знает, где именно хранятся данные – в массиве или базе данных, – но знает, что он получит данные от адаптера. Соответственно, он обращается к адаптеру и запрашивает первых пять записей.

Объект *CursorAdapter* получает курсор при создании, но обращается к нему за данными только тогда, когда потребуется.

Несмотря на то что таблица базы данных содержит много записей, курсор должен прочитать только первых пять. Такой подход эффективнее – данные появятся на экране намного быстрее.

Когда пользователь прокручивает список, *CursorAdapter* приказывает курсору прочитать другие записи из базы данных. Если пользователь ограничивается минимальной прокруткой и открывает только одну новую строку, курсор читает из базы данных одну запись.

Класс *SimpleCursorAdapter* – специализация *CursorAdapter*, которая может использоваться в большинстве ситуаций с выводом данных курсора в списковом представлении. Он получает столбцы из курсора и связывает их с компонентом *TextViews* или *ImageViews.*

Первое, что необходимо учесть при создании курсора для адаптера, – какие столбцы должен содержать курсор. В курсор следует включить все столбцы, которые должны отображаться в списковом представлении, а также столбец с именем *_id*. Этот столбец должен быть включен обязательно, иначе адаптер курсора работать не будет.

При создании простого адаптера следует указать, как должны выводиться данные, какой курсор следует использовать и какие столбцы должны связываться с теми или иными представлениями:

```
CursorAdapter listAdapter = new SimpleCursorAdapter(this,
        android.R.layout.simple_list_item_1,
        cursor,
        new String[]{DBEntry.COLUMN_NAME_NAME}
        new int[]{android.R.id.text1},
        0);

listStudent.setAdapter(listAdapter);
```

Как и в случае с адаптером массива, здесь используется *android.R.layout.simple_list_item_1*, чтобы сообщить, что каждая строка в курсоре должна отображаться в виде одного текстового *view* в списковом представлении. Этому текстовому представлению назначен идентификатор *android.R.id.text1*.

Первый и второй параметры конструктора, *context* и *layout*, те же, что использовались при создании адаптера массива. В параметре *context* (*this*) передается текущий контекст. Вместо массива, из которого должны загружаться данные, нужно задать курсор с данными при помощи параметра *cursor*. Четвертый параметр указывает, какие столбцы использовать в курсоре, а пятый – в каких представлениях они

должны отображаться. В последнем параметре обычно передается 0 (значение по умолчанию). Также можно передать значение *FLAG_REGISTER_CONTENT_OBSERVER* для регистрации наблюдателя, который будет оповещаться об изменении данных. Здесь эта возможность не рассматривается.

Курсоры не отслеживают изменения информации в базе данных. Если информация изменится после создания курсора, то он обновлен не будет. Он по-прежнему будет содержать исходные записи без каких-либо изменений:

```
public void onRestart() {
    super.onRestart();
...
db =
Cursor cursor =
ListView list =
CursorAdapter =
//Замена курсора, используемого CursorAdapter, только что созданным
adapter.changeCursor(cursor);
```

Курсор должен оставаться открытым на тот случай, если из него потребуется прочитать дополнительные данные. Это означает, что курсор и базу данных не удастся закрыть сразу же после вызова метода *setAdapter*() для связывания со списковым представлением. Вместо этого для закрытия можно воспользоваться методом *onDestroy*() активности. Так как активность готовится к уничтожению, сохранять связь с курсором или базой данных не нужно, их можно закрыть:

```
public void onDestroy(){
    super.onDestroy();
    cursor.close();
    db.close();
}
```

10.6 Работа с потоками. Класс AsyncTask

Основная проблема с обращениями к базе данных заключается в том, что она может замедлить реакцию приложения на действия пользователя. Начиная с версии Lollipop, существуют три вида потоков, которые необходимо учитывать:

1) ***основной поток событий.*** Этот поток прослушивает интенты, получает сообщения о касаниях от экрана и вызывает все методы внутри ваших активностей;

2) ***поток визуализации.*** Данный поток (обычно пользователь не взаимодействует с ним) читает список запросов на обновление экрана, а затем выдает команды низкоуровневому графическому оборудованию на его перерисовку;

3) ***все остальные потоки.***

Приложение выполняет почти всю свою работу в основном потоке событий. Если включить код базы данных в метод *onCreate*(), то основной поток событий будет занят работой с базой данных вместо того, чтобы заниматься обработкой событий от экрана или других приложений.

Таким образом, необходимо ***вынести код базы данных из основного потока событий*** и выполнить его в отдельном потоке в фоновом режиме. Подготовку интерфейса, загрузку *view* и другую работу оставим в основном потоке.

Код обновления представлений из базы данных обязательно выполняется в главном потоке, иначе произойдет исключение. Для этого существует ряд классов: *Executor*, *ThreadPoolExecutor* и *FutureTask* используют для длительных операций; *AsyncTask* – помощник над *Thread*; *Handler* – подходит для выполнения небольших операций в фоновом режиме и позволяет обновлять представления в основном потоке событий, когда операции завершаются.

Чтобы создать свою реализацию *AsyncTask*, следует расширить класс *AsyncTask* и реализовать метод *doInBackground*(). Код этого метода выполняется в фоновом потоке, поэтому он подходит для размещения кода базы данных. *AsyncTask* также содержит методы *onPreExecute*() и *onPostExecute*(), которые выполняются до и после метода *doInBackground*(). Также имеется метод *onProgressUpdate*() на тот случай, если необходимо передать информацию о ходе выполнения задачи:

```
private class SomeTask extends AsyncTask<Params, Progress, Result>    {

        protected void onPreExecute() {
    //Код, предшествующий выполнению задачи
        }

          protected Result doInBackground(Params... params) {
    //Код, выполняемый в фоновом потоке
         }

         protected void onProgressUpdate(Progress... values) {
    //Код, передающий информацию о ходе выполнения задачи
         }

         protected void onPostExecute(Result result) {
    //Код, выполняемый при завершении задачи
        }
}
```

AsyncTask определяется тремя обобщенными параметрами: *Params*, *Progress* и *Results*:

— *Params* – тип объекта, используемый для передачи произвольных параметров задачи методу *doInBackground*();

— *Progress* – тип объекта, используемый для передачи информации о прогрессе задачи, тип параметров *onProgressUpdate*();

— *Result* – тип результата задачи, тип параметра *onPostExecute*().

Если любые из этих параметров не используются, в них можно передать *Void*.

10.6.1. Метод onPreExecute()

Этот метод вызывается до начала фоновой задачи и используется для подготовки ее выполнения. Метод вызывается в основном потоке событий, поэтому для него доступны все представления в пользовательском интерфейсе. *onPreExecute*() вызывается без параметров и возвращает *void*.

```
private class SomeTask extends AsyncTask<Params, Progress, Result> {

    ContentValues newValues;

    protected void onPreExecute() {

        CheckBox choose = (CheckBox)findViewById(R.id.choose);
        newValues = new ContentValues();
        newValues.put("CHOOSE", choose.isChecked());

    }
```

10.6.2. Метод doInBackground()

Метод *doInBackground*() запускается в фоновом режиме сразу же после выполнения *onPreExecute*(). Определяет тип передаваемых задаче параметров и тип возвращаемого значения. Метод можно использовать для кода работы с базой данных, чтобы он выполнялся в фоновом потоке.

Метод получает идентификатор из списка, информацию о котором требуется обновить, а логическое возвращаемое значение позволит проверить, успешно ли была выполнена задача:

```
protected Boolean doInBackground(Integer... chooses) {

    int chooseNo = chooses[0];

    DbHelper mDbHelper = new DbHelper(getContext());
```

```
    try {
        SQLiteDatabase db = mDbHelper.getWritableDatabase();
        db.update(DBContract.DBEntry.TABLE_NAME, newValues,
                "_id = ?", new String[] {Integer.toString(chooseNo)});
        db.close();
        return true;
    }
    catch(SQLiteException e) {
        return false;
    }
}
```

10.6.3. Метод onProgressUpdate()

Метод *onProgressUpdate*() вызывается в основном потоке событий, поэтому в нем доступны представления пользовательского интерфейса. Метод может использоваться для вывода сведений о ходе выполнения операции. Определяет тип параметров, которые должны передаваться методу.

В примере с базами данных информация о ходе выполнения задачи не публикуется, поэтому реализовывать этот метод не нужно.

10.6.4. Метод onPostExecute()

Метод *onPostExecute*() вызывается после завершения фоновой задачи в основном потоке событий, а следовательно, для него доступны все представления в пользовательском интерфейсе. Метод может использоваться для отображения результатов задачи. Ему передаются результаты метода *doInBackground*(), поэтому его параметры должны соответствовать возвращаемому типу *doInBackground*(). Будем использовать метод *onPostExecute*() для проверки того, успешно ли был выполнен код базы данных в методе *doInBackground*(). Если при выполнении произошла ошибка, приложение выводит сообщение для пользователя.

```
private class SomeTask extends
                        AsyncTask<Integer, Void, Boolean> {

    protected void onPostExecute(Boolean success) {
        if (!success) {
            Toast toast = Toast.makeText(MianActivity.this,
            "Database unavailable", Toast.LENGTH_SHORT);
            toast.show();
        }
    }
```

10.6.5. Выполнение задачи

Чтобы запустить задачу на выполнение, вызывается метод *execute*() объекта *AsyncTask*. Если метод *doInBackground*() получает параметры, они добавляются в метод *execute*():

```
public class MainActivity extends AppCompatActivity {

      private         SomeTask dbtask;

@Override
    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dbtask = new SomeTask();
        dbtask.execute();
...
```

Например, методу *doInBackground*() задачи *AsyncTask* должен передаваться номер, выбранный пользователем, поэтому вызов выглядит так:

```
new SomeTask().execute(chooses);
```

Тип параметра *execute*() должен соответствовать типу параметра, который ожидает получить метод *doInBackground*() объекта *AsyncTask*.

Возвращение данных из *AsyncTask* выполняется методом *get*. Метод *get* просто блокирует поток, в котором он выполняется, и не отпускает до тех пор, пока не будет получен какой-то результат или не произойдет *Exception*.

В примере с базами данных *onPostExecute* ничего не возвращает, но если надо было бы вернуть результат, то вернуть можно так:

```
new SomeTask().execute(chooses).get();
```

Есть реализация метода *get* с таймаутом. В этом случае *get* будет ждать указанное время, а потом сгенерирует *Exception*. Если же задача уже была завершена, то метод выполнится сразу:

```
try {
    String result2 =   dbtask.execute(chooses)
                              .get(2000, TimeUnit.MILLISECONDS);
}
catch (TimeoutException e) {
    e.printStackTrace();
}
```

10.6.6. Отмена задачи

Иногда возникает необходимость отменить уже выполняющуюся задачу. Для этого в *AsyncTask* есть метод *cancel*. Он на вход принимает *boolean*-параметр, который указывает, может ли система прервать выполнение потока.

В *doInBackground* нужно периодически вызывать метод *isCancelled*. Как только выполнится метод *cancel* для *AsyncTask*, *isCancelled* будет возвращать *true*. А это значит, что нужно завершить метод *doInBackground*.

Таким образом, метод *cancel* – это маркер того, что задачу надо отменить. Метод *isCancelled* читает метку и предпринимает действия для завершения работы задачи. Метод действует менее жестко и просто возвращает *true* при вызове *isCancelled*().

```
dbtask.cancel(false);
```

Метод *doInBackground* продолжает работу; метод *onPostExecute* не выполняется.

```
dbtask.cancel(true);
```

Метод *doInBackground* завершает работу, происходит *InterruptedException* (но может и не произойти).

10.6.7. Статусы задачи

Всегда можно определить, в каком состоянии сейчас находится задача. Для этого используются статусы:

- *PENDING* – задача еще не запущена;
- *RUNNING* – задача в работе;
- *FINISHED* – метод *onPostExecute*() отработал, т. е. задача успешно завершена.

```
private void showStatus() {
    if (dbtask != null)
        Toast.makeText(this, dbtask.getStatus().toString(),
Toast.LENGTH_SHORT).show();
}
```

10.6.8. Поворот экрана

Ранее в примерах для *Activity* создавали внутренний класс, наследующий *AsyncTask*, потом создавали экземпляр этого класса и работали с ним. Но, если повернуть экран, *Activity* будет создано заново, все прошлые объекты будут потеряны. В том числе и ссылка на *AsyncTask*. Сам *AsyncTask* будет работать со старым *Activity* и держать его в памяти, так как объект внутреннего класса (*AsyncTask*) содержит скрытую ссылку на объект внешнего класса (*Activity*).

Это значит, что при повороте экрана продолжит работу старый *Task*, который работает со старым *MainActivity*. А параллельно с ним будет работать новый *Task* с новым *MainActivity*, и работа начнется сначала. Старые объекты продолжают существовать где-то в памяти и работать.

Каждый раз начинать задачу заново при повороте экрана – это плохо, поэтому надо при создании нового *Activity* получать ссылку на старый *Task* и не создавать новый. В этом могут помочь методы *onRetainNonConfigurationInstance* и *getLastNonConfigurationInstance*.

Для этого следует добавить в класс *MainActivity* реализацию метода *onRetainNonConfigurationInstance*:

```
public Object onRetainNonConfigurationInstance() {

    return dbtask;
}
```

При повороте экрана система сохранит ссылку на объект *dbtask* в методе *onCreate*:

```
public void onCreate(Bundle savedInstanceState) {

    //...
    dbtask = (SomeTask) getLastNonConfigurationInstance();
    if ( dbtask == null) {
        dbtask = new SomeTask();
        dbtask.execute();
    }
...
```

При создании *Activity* системе посылается запрос вернуть (*getLastNonConfigurationInstance*) сохраненный в методе *onRetainNonConfigurationInstance* объект и привести его к *SomeTask*. Если *Activity* создается до поворота экрана, то в ответ приходит *null*, а значит, создается *SomeTask*.

Работать со старыми ссылками плохо, потому что старый *SomeTask* связан со своим объектом внешнего класса *MainActivity* и видит только его, а также изменяет старый *MainActivity*, который висит в памяти. На экране виден новый *MainActivity*. И он не меняется.

Чтобы создать ссылку на *MainActivity*, надо в *SomeTask* описать объект, который будет на него ссылаться на *MainActivity*. Когда создается новая активность, ссылка на него будет передаваться в *SomeTask*:

```
public void onCreate(Bundle savedInstanceState) {

        //...
    dbtask = (SomeTask) getLastNonConfigurationInstance();
```

```
            if ( dbtask == null) {
                dbtask = new SomeTask();
                dbtask.execute();
            }
 // Передаем в SomeTask ссылку на текущее MainActivity
            dbtask.link(this);

public Object onRetainNonConfigurationInstance() {

    // Удаляем из SomeTask ссылку на старое MainActivity
    dbtask.unLink();
    return dbtask;
}

static class SomeTask extends  AsyncTask<Integer, Void, Boolean> {

    MainActivity activity;

    // Получаем ссылку на MainActivity
    void link(MainActivity act) {
        activity = act;
    }

    // Обнуляем ссылку
    void unLink() {
        activity = null;
    }
}
```

Здесь добавляется *static* к описанию класса *SomeTask*. Также описываются объект *activity* класса *MainActivity* и два метода:

— *link* – с его помощью *SomeTask* будет получать ссылку на *MainActivity* для работы;

— *unlink* – обнуление ссылки.

Теперь в классе *SomeTask* невозможно просто так работать с объектами *MainActivity*, так как *SomeTask* – *static*, он не содержит скрытой ссылки на *MainActivity*.

В методе *onRetainNonConfigurationInstance* перед тем, как сохранить *SomeTask* для передачи новому *Activity*, обнуляется ссылка на старый *MainActivity*. *SomeTask* больше не будет держать старый *MainActivity*, и система сможет его уничтожить.

В *onCreate* после создания/получения объекта *SomeTask* вызывается метод *link* и передается туда ссылку на текущий новый *MainActivity*. С ним и продолжается работа *SomeTask*.

Таким образом, код работы с базой данных всегда должен выполняться в фоновом режиме, метод *doInBackground* реализуется обязательно, объект *AsyncTask* создается в UI-потоке, а метод *execute* вызывается в этом потоке, методы *onPreExecute*, *doInBackground*, *onPostExecute* нельзя вызывать напрямую. Метод *AsyncTask* может быть запущен (*execute*) только один раз, иначе будет получено исключение.

11. ROOM И LIVEDATA. АРХИТЕКТУРА ПРИЛОЖЕНИЯ

В мае 2018 г. на Google был представлен I/O Android Jetpack. Это небольшие разделяемые библиотеки, которые не являются частью базовой платформы Android, это своего рода набор инструментов, рекомендуемый для использования Google (рис. 11.1). Все библиотеки Android Jetpack были перемещены в новое пространство имен androidx.*.

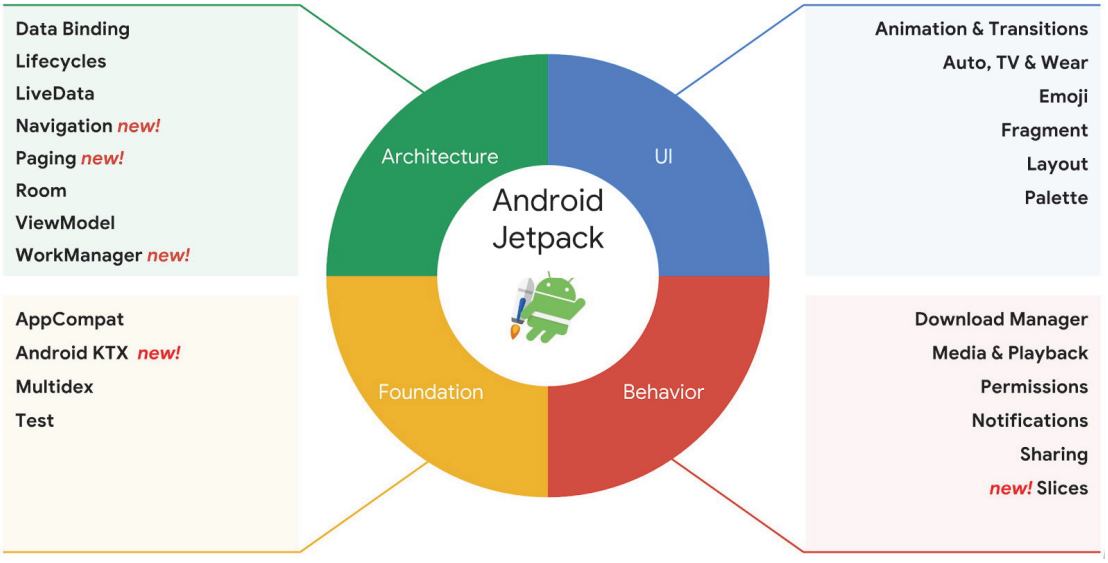


Рис. 11.1. Структура Android Jetpack

Изначально в Android было два основных метода для выполнения фоновой работы: фоновый поток (*AsyncTask*, например) и сервисы. Среди компонентов Jetpack оказался новый компонент – *WorkManager*. Это работающая на основе существующего Android библиотека, которая позволяет делать любые фоновые задания с реактивным программированием в нужное время, нужной последовательности и нужных условиях:

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

Компонент *Paging* 1.0.0 позволяет легко загружать и представлять большие наборы данных с быстрой и бесконечной прокруткой в *RecyclerView*. Он может загружать выгружаемые данные из локального хранилища, сети или и того, и другого, а также позволяет определять, как загружается контент.

В пакет также были добавлены следующие элементы: *Slices* – способ разместить пользовательский интерфейс приложения внутри Google Assistant в результате поиска, *Android KTX* – для использования возможностей языка Kotlin и др.

11.1. Компоненты архитектуры

Компоненты архитектуры помогают структурировать приложение и обеспечить поддержку с меньшим количеством шаблонов.

Архитектура Android приложения фокусируется на подмножестве компонентов, а именно *LiveData*, *ViewModel* и *Room* (рис. 11.2).

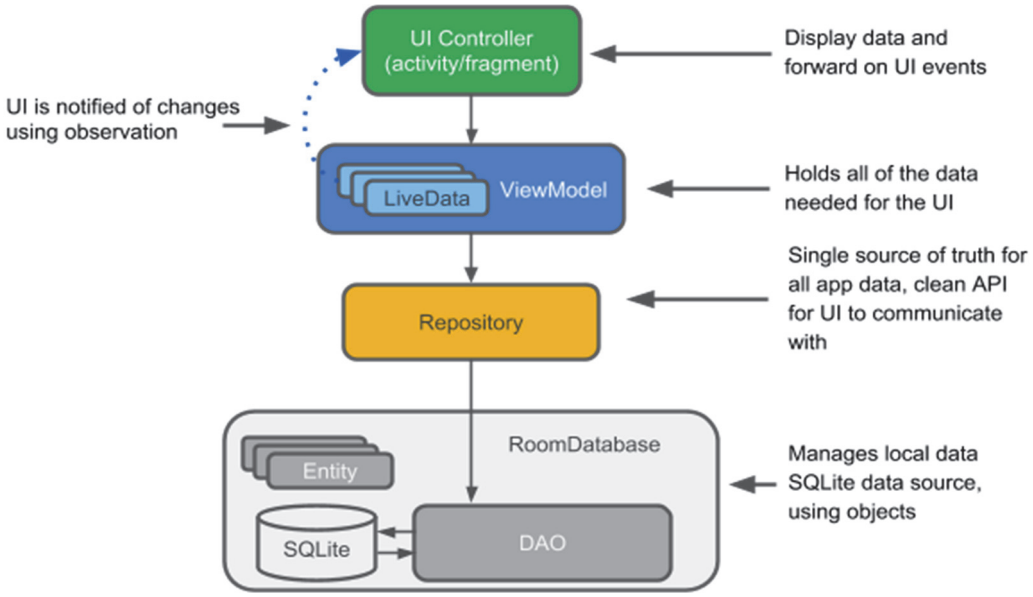


Рис. 11.2. Архитектура приложения

11.2. Lifecycle

Часть логики приложения часто завязана на жизненный цикл *Activity*. Ниже приведен пример соединения с сервером в методах *onStart* или *onResume* и отключения от сервера в *onPause* или *onStop*:

```
public class MyServer {
    public void connect() {
        // ...
    }
    public void disconnect() {
        // ...
    }
}

@Override
protected void onStart() {
      super.onStart();
      myServer.connect();
      }

@Override
protected void onStop() {
      super.onStop();
      myServer.disconnect();
      }
```

Это часто используемая схема. Но в сложных приложениях содержание методов *onStart*, *onStop* и пр. может быть достаточно запутанным. Для улучшения архитектуры Google рекомендует выносить эту логику из активности. Это можно сделать так. У *Activity* есть метод *getLifecycle*, который возвращает объект *Lifecycle*. На этот объект можно подписать слушателей, которые будут получать уведомления при смене *lifecycle*-состояния активности.

Активности и фрагменты в *Support Library* начиная с версии 26.1.0 реализуют интерфейс *LifecycleOwner*. Этот интерфейс добавляет им метод *getLifecycle*.

В *build.gradle* в секции *dependencies* должна быть такая зависимость:

```
implementation 'com.android.support:appcompat-v7:26.1.0'
```

В примере слушателем будет *MyServer* и он должен наследовать интерфейс *LifecycleObserver*:

```
public class MyServer implements LifecycleObserver {

    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    public void connect() {
        // ...
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
    public void disconnect() {
        // ...
    }
}
```

В классе *MyServer* методы помечаются аннотацией *OnLifecycle-Event* и указывается, при каком *lifecycle*-событии метод должен быть вызван.

В активности методом *getLifecycle* получается объект типа *Lifecycle* и методом *addObserver* подписывается *myServer* (методы *onStart* и *onStop* больше не нужны):

```
@Override
protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      // ...
      getLifecycle().addObserver(myServer);
      }
```

При переходе *Activity* из состояния *CREATED* в состояние *STARTED* его объект *Lifecycle* вызовет метод *myServer.connect*. А при переходе из *STARTED* в *CREATED* вызывается *myServer.disconnect* (рис. 11.3).

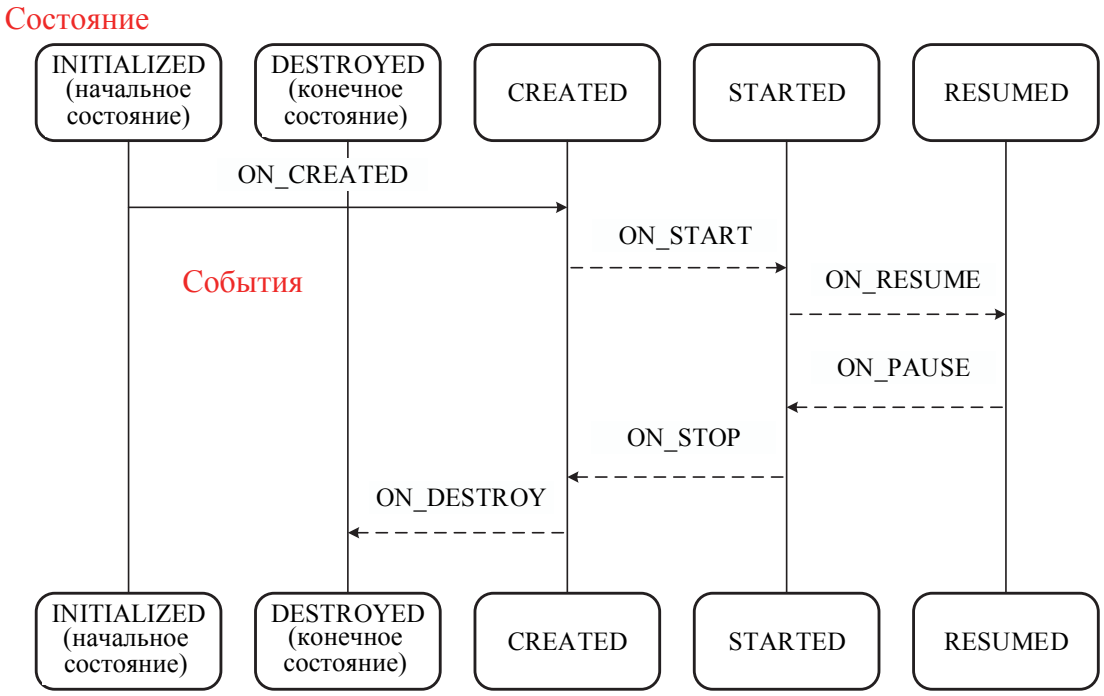


Рис. 11.3. Переходы между состояниями *Activity* и вызов методов

Отписаться от *Lifecycle* можно методом *removeObserver*().

При этом можно использовать событие *ON_ANY* для получения всех событий в одном методе:

```
@OnLifecycleEvent(ON_ANY)
void onAny(LifecycleOwner source, Lifecycle.Event event) {
      // ...
      }
```

Чтобы узнать текущее состояние активности, нужно вызвать метод *getCurrentState* ():

```
if (getLifecycle().getCurrentState() == Lifecycle.State.RESUMED) {
      // ...
      }
```

Метод *isAtLeast*() проверяет состояние активности, которое должно быть не ниже, чем *STARTED*, то есть либо *STARTED*, либо *RESUMED*:

```
if (getLifecycle().getCurrentState().isAtLeast(Lifecy-
cle.State.STARTED)) {
      // ...
      }
```

11.3. LiveData

В файле *build.gradle* модуля следует добавить зависимости:

```
dependencies {
    implementation "android.arch.lifecycle:extensions:1.0.0"
    annotationProcessor "android.arch.lifecycle:compiler:1.0.0"
    ...
}
```

LiveData – хранилище данных, работающее по принципу паттерна *Observer* (наблюдатель). В него можно поместить какой-либо объект и на него можно подписаться и получать объекты, которые в него помещают (аналогия – каналы в Telegram: автор пишет пост и отправляет его в канал, а все подписчики получают этот пост).

LiveData умеет определять, активен подписчик или нет, и ***отправляет данные только активным подписчикам***. Подписчиками *LiveData* будут активности и фрагменты. А их состояние будет определяться с помощью *Lifecycle* объекта.

11.3.1. Получение данных из LiveData

Пусть имеется синглтон – класс *DataController*, из которого можно получить *LiveData<String>*.

```
LiveData<String> liveData = DataController.getInstance().getData();
```

DataController периодически работает и обновляет данные в *LiveData*. *Activity* может подписаться на *LiveData* и получать данные, которые помещаются в его *DataController*:

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

   LiveData<String> liveData = DataController.getInstance().getData();

liveData.observe(this, new Observer<String>() {
  @Override
  public void onChanged(@Nullable String value) {
      textView.setText(value)
      }
   });
}
```

Получают *LiveData* из *DataController* и подписываются методом *observe*(). В метод *observe*() необходимо передать два параметра: первый – *LifecycleOwner* (*LiveData* получит из *Activity* его *Lifecycle* и по нему будет определять состояние *Activity*; например, когда *Activity* видна на экране, *LiveData* будет считать ее активной и отправлять данные в ее *callback*); второй параметр – это непосредственно подписчик, то есть *callback*, в который *LiveData* будет отправлять данные. В нем только один метод *onChanged*().

Теперь, когда *DataController* поместит какой-либо *String*-объект в *LiveData*, сразу можно получить этот объект в активности, если она находится в состоянии *STARTED* или *RESUMED* (если же она была не активна во время обновления данных в *LiveData*, то при возврате в активное состояние ее *observer* получит последнее актуальное значение данных).

Если активность будет закрыта, то есть перейдет в статус *DESTROYED*, то *LiveData* автоматически отпишет от себя *observer*. Если активность в состоянии *DESTROYED* попробует подписаться, то подписка не будет выполнена.

Если активность уже подписывала свой *observer* и попробует сделать это еще раз, то ничего не произойдет. Всегда можно получить последнее значение *LiveData* с помощью метода *getValue*().

Поворот экрана и полное закрытие активности обрабатывается автоматически.

11.3.2. Отправка данных в LiveData

В классе *DataController* переменная *LiveData* будет определена так:

```
private MutableLiveData<String> liveData = new MutableLiveData<>();

      LiveData<String> getData() {
      return liveData;
      }
```

LiveData позволяет внешним объектам только получать данные. Но внутри *DataController* используется объект *MutableLiveData*, который позволяет помещать в него данные. Чтобы поместить значение в *MutableLiveData*, используется метод *setValue*:

```
liveData.setValue("new value");
```

Этот метод обновит значение *LiveData*, и все его активные подписчики получат это обновление. Метод *setValue*() должен быть вызван из UI-потока. Для обновления данных из других потоков следует использовать метод *postValue*(). Он перенаправит вызов в UI-поток. Соответственно, подписчики всегда будут получать значения в основном потоке.

11.3.3. Transformations

Можно поменять тип данных в *LiveData* с помощью *Transformations.map.*

Ниже представлен пример, в котором *LiveData<String>* будет преобразовываться в *LiveData<Integer>*:

```
LiveData<String> liveData = ...;

      LiveData<Integer> liveDataInt = Transformations.map(liveData,
                                    new Function<String, Integer>() {
@Override
public Integer apply(String input) {
      return Integer.parseInt(input);
      }
 });
```

В метод *map* передается имеющийся *LiveData<String>* и функция преобразования. В этой функции получают *String*-данные из *LiveData<String>* и конвертируют их в *Integer*. На выходе метода *map* получим *LiveData<Integer>*.

Рассмотрим более сложный случай. Например, имеется *LiveData<Long>*, необходимо получить из него *LiveData<User>*. Конвертация *id* в *User* выглядит так:

```
private LiveData<User> getUser(long id) {
      // ...
      }
```

По *id* получают *LiveData<User>* и на него надо будет подписываться, чтобы получить объект *User*. Использовать метод *map* невозможно,

так как на выходе будет объект *LiveData<LiveData<User>>*. Чтобы избежать этого, надо использовать *switchMap* вместо *map*:

```
LiveData<Long> liveDataId = ...;

LiveData<User> liveDataUser = Transformations.switchMap(liveDataId,
                              new Function<Long, LiveData<User>>() {
@Override
public LiveData<User> apply(Long id) {
      return getUser(id);
   }
});
```

11.3.4. Пользовательский тип LiveData

В некоторых ситуациях удобно создать свою обертку *LiveData*. Например:

```
public class LocationLiveData extends LiveData<Location> {

   LocationService.LocationListener locationListener =
         new LocationService.LocationListener() {
            @Override
            public void onLocationChanged(Location location) {
               setValue(location);
            }
         };

   @Override
   protected void onActive() {
      LocationService.addListener(locationListener);
   }

   @Override
   protected void onInactive() {
      LocationService.removeListener(locationListener);
   }
}
```

Класс *LocationLiveData* расширяет *LiveData<Location>*. У него есть *locationListener* – слушатель, который можно передать в *LocationService* и получать обновления текущего местоположения. При получении нового *Location* от *LocationService locationListener* будет вызывать метод *setValue* и тем самым обновлять данные этого *LiveData*.

LocationService – это сервис, который предоставляет нам текущую локацию. Его реализация в данном примере не важна. Главное то, что можно подписаться (*addListener*) на сервис, когда нужны данные, и отписаться (*removeListener*), когда данные больше не нужны.

Переопределены методы *onActive* (будет вызван, когда у *LiveData* появится хотя бы один подписчик) и *onInactive* (когда не останется

ни одного подписчика). Это удобная обертка, которая при появлении подписчиков сама будет подписываться к *LocationService*, получать *Location* и передавать его своим подписчикам. А когда подписчиков не останется, то *LocationLiveData* отпишется от *LocationService*.

11.3.5. MediatorLiveData

MediatorLiveData дает возможность собирать данные из нескольких *LiveData* в один. Это удобно, если имеется несколько источников. Следует объединить их в один и подписаться только на него.

Рассмотрим, как это делается, на простом примере.

```
MutableLiveData<String> liveData1 = new MutableLiveData<>();
MutableLiveData<String> liveData2 = new MutableLiveData<>();

MediatorLiveData<String> mediatorLiveData = new MediatorLiveData<>();

mediatorLiveData.addSource(liveData1, new Observer<String>() {
@Override
public void onChanged(@Nullable String s) {
      mediatorLiveData.setValue(s);
       }
});

 mediatorLiveData.addSource(liveData2, new Observer<String>() {
@Override
public void onChanged(@Nullable String s) {
      mediatorLiveData.setValue(s);
     }
});
```

Метод *addSource*() требует два параметра: *LiveData*, из которого *MediatorLiveData* собирается получать данные; *callback*, который будет использован для подписки на *LiveData* из первого параметра. В методе *callback* надо самим передавать в *MediatorLiveData* данные, получаемые из *LiveData*. Это делается методом *setValue*().

Таким образом *mediatorLiveData* будет получать данные из двух *LiveData* и передавать их своим получателям.

Если подписаться на *mediatorLiveData*:

```
mediatorLiveData.observe(this, new Observer<String>() {
@Override
public void onChanged(@Nullable String s) {
      log("onChanged " + s);
  }
});
```

то сюда будут приходить данные из *liveData1* и *liveData2*.

Если надо отписаться от *liveData2*, то это будет выглядеть так:

```
mediatorLiveData.addSource(liveData1, new Observer<String>() {
@Override
public void onChanged(@Nullable String s) {
      mediatorLiveData.setValue(s);
   }
});

mediatorLiveData.addSource(liveData2, new Observer<String>() {
@Override
public void onChanged(@Nullable String s) {
      if ("finish".equalsIgnoreCase(s)) {
         mediatorLiveData.removeSource(liveData2);
         return;
      }
      mediatorLiveData.setValue(s);
   }

});
```

В случае с *liveData1* ничего не меняется. Но при получении данных от *liveData2* надо смотреть на значение: если это “finish”, то методом *removeSource* следует отписаться в *mediatorLiveData* от *liveData2* и не передавать это значение дальше.

11.3.6. RxJava

Можно конвертировать *LiveData* в *Rx* и наоборот. Для этого есть инструмент *LiveDataReactiveStreams*.

Нужно добавить зависимости:

```
implementation "android.arch.lifecycle:reactivestreams:1.0.0"
```

Чтобы получить *LiveData* из *Flowable* или *Observable*, надо использовать метод *fromPublisher*():

```
Flowable<String> flowable = ... ;
LiveData<String> liveData = LiveDataReac-
tiveStreams.fromPublisher(flowable);
```

LiveData будет подписан на *Flowable*, пока у него (у *LiveData*) есть подписчики. *LiveData* не сможет обработать или получить *onError*. Неважно, в каком потоке работает *Flowable*, результат в *LiveData* всегда придет в UI-потоке.

Чтобы получить *Flowable* или *Observable* из *LiveData*, нужно выполнить два преобразования. Сначала следует использовать метод

toPublisher(), чтобы получить *Publisher*. Затем полученный *Publisher* надо передать в метод *Flowable.fromPublisher*():

```
LiveData<String> liveData = … ;
Flowable<String> flowable = Flowable.fromPublisher(
      LiveDataReactiveStreams.toPublisher(this, liveData));
```

Прочие методы *LiveData*:

— *hasActiveObservers*() – проверка наличия активных подписчиков;

— *hasObservers*() – проверка наличия любых подписчиков;

— *observeForever* (*Observer*<T> *observer*) – позволяет подписаться без учета *Lifecycle*. То есть этот подписчик будет всегда считаться активным;

— *removeObserver* (*Observer*<T> *observer*) – дает возможнось отписать подписчика;

— *removeObservers* (*LifecycleOwner owner*) – позволяет отписать всех подписчиков, которые завязаны на *Lifecycle*, от указанного *LifecycleOwner*.

11.4. ViewModel

ViewModel – класс, позволяющий активностям и фрагментам сохранять необходимые им объекты «живыми» при повороте экрана.

Создаем свой класс, наследующий *ViewModel:*

```
public class MyViewModel extends ViewModel {

}
```

Чтобы добраться до него в активности, нужен следующий код:

```
@Override
protected void onCreate(Bundle savedInstanceState) {

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

   MyViewModel model = ViewModelProviders.of(this).get(MyView-
Model.class);
   // ...
}
```

В метод *ViewModelProviders.of* необходимо передать активность. При этом получают доступ к провайдеру, который хранит все *ViewModel* для этой *Activity*. Методом *get*() запрашивают у провайдера конкретную

модель по имени класса, например *MyViewModel*. Если провайдер еще не создавал такой объект ранее, то он его создает и возвращает. И пока активность окончательно не закрыта, при всех последующих вызовах метода *get*() будем получать этот же объект *MyViewModel*.

Соответственно, при поворотах экрана активность будет пересоздаваться, а объект *MyViewModel* будет спокойно себе «жить» в провайдере. Активность после пересоздания сможет получить этот объект обратно и продолжить работу.

Следовательно, не нужно хранить в *ViewModel* ссылки на активности, фрагменты, View и пр. Это может привести к утечкам памяти.

На рис. 11.4 показано время жизни (scope) модели. Модель «жива», пока активность не закроется окончательно. Можно создавать несколько моделей одного и того же класса, но использовать разные текстовые ключи для их хранения в провайдере.

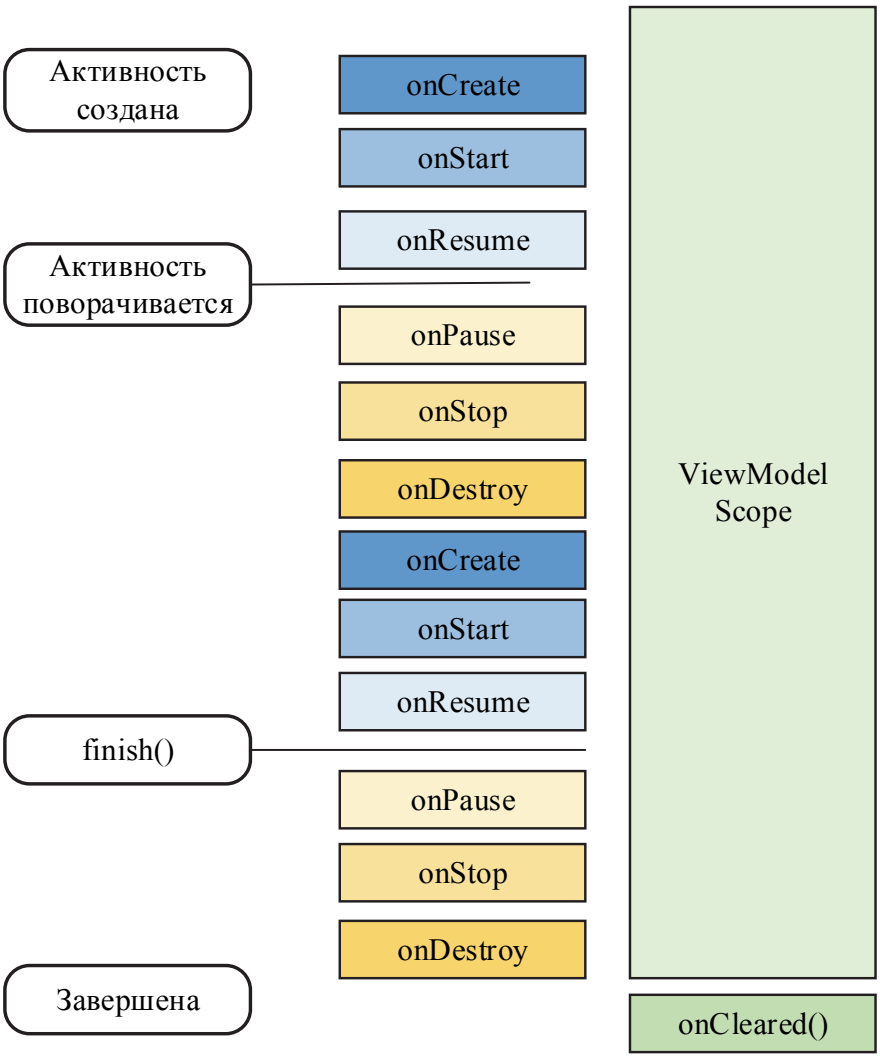


Рис. 11.4. Время жизни *ViewModel*

11.4.1. LiveData и ViewModel

LiveData удобно использовать с *ViewModel*.

Например, асинхронная однократная загрузка каких-либо данных:

```
public class MyViewModel extends ViewModel {
    // ...
    MutableLiveData<String> data;
    public LiveData<String> getData() {
        if (data == null) {
            data = new MutableLiveData<>();
            loadData();
        }
        return data;
    }
    private void loadData() {
        dataRepository.loadData(new Callback<String>() {
            @Override
            public void onLoad(String s) {
                data.postValue(s);
            }
        });
    }

}
```

Основной метод – *getData*. Когда активность захочет получить данные, она вызовет этот метод. Следует проверить, создан ли *MutableLiveData*. Если нет, значит этот метод вызывается первый раз. В таком случае надо создать *MutableLiveData* и запустить асинхронный процесс получения данных методом *loadData*(). Далее нужно вернуть *LiveData*.

Метод *loadData*() должен быть асинхронным, потому что он вызывается из метода *getData*, а *getData* в свою очередь вызывается из активности и все это происходит в UI-потоке. Если *loadData*() начнет загружать данные синхронно, то это заблокирует UI-поток.

Код в *Activity* выглядит так:

```
@Override
protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
MyViewModel model = ViewModelProviders.of(this)
                                        .get(MyViewModel.class);
    LiveData<String> data = model.getData();
    data.observe(this, new Observer<String>() {
        @Override
        public void onChanged(@Nullable String s) {
            // ...
        }
    });
}
```

От провайдера необходимо получить модель, а от модели – *LiveData*, на который надо подписаться и подождать данных. *ViewModel* нужен, чтобы сохранить процесс получения данных при повороте экрана, а *LiveData* – для удобного асинхронного получения данных.

Последовательность действий выглядит так:

— активность вызывает метод модели *getData*;

— модель создает *MutableLiveData* и запускает асинхронный процесс получения данных от репозитория;

— активность подписывается на полученный от модели *LiveData* и ждет данные;

— происходит поворот экрана;

— на модели этот поворот никак не сказывается, она расположена в провайдере и ждет ответ от репозитория;

— активность пересоздается, получает ту же самую модель от провайдера, получает тот же самый *LiveData* от модели, подписывается на него и ждет данные;

— репозиторий возвращает данные, модель передает их в *MutableLiveData*;

— активность получает данные от *LiveData*.

Если репозиторий вдруг пришлет ответ в тот момент, когда *Activity* будет пересоздаваться, то *Activity* получит этот ответ, как только подпишется на *LiveData*.

Если репозиторий сам умеет возвращать *LiveData*, то все значительно упрощается. Нужно просто отдать *LiveData* в активность и подписать его.

```
public class MyViewModel extends ViewModel {

    // ...

    LiveData<String> data;

    public LiveData<String> getData() {
        if (data == null) {
            data = dataRepository.loadData();
        }
        return data;
    }
}
```

11.4.2. Очистка ресурсов

Когда активность окончательно закрывается, провайдер удаляет *ViewModel*, предварительно вызвав метод *onCleared*():

```
public class MyViewModel extends ViewModel {
    // ...
    @Override
    protected void onCleared() {
        // clean up resources
    }
}
```

11.4.3 Context

Не стоит передавать активность в модель в качестве *Context*. Это может привести к утечкам памяти. Если в модели понадобится объект *Context*, то можно наследовать не *ViewModel*, а *AndroidViewModel*.

```
public class MyViewModel extends AndroidViewModel {

    public MyViewModel(@NonNull Application application) {
        super(application);
    }

    public void doSomething() {
        Context context = getApplication();
        // ....
    }

}
```

При создании этой модели провайдер передаст ей в конструктор класс *Application*, который является *Context*. Его можно получить методом *getApplication*().

11.4.4. Передача данных между фрагментами

ViewModel может быть использована для передачи данных между фрагментами, которые находятся в одной активности. В официальной документации есть такой пример кода:

```
public class SharedViewModel extends ViewModel {
    private final MutableLiveData<Item> selected =
                                    new MutableLiveData<Item>();

    public void select(Item item) {
        selected.setValue(item);
    }

    public LiveData<Item> getSelected() {
        return selected;
    }
}
```

SharedViewModel – модель с двумя методами: один позволяет поместить данные в *LiveData*, другой – получить сам *LiveData*. Соответственно, если два фрагмента будут иметь доступ к этой модели, то

один сможет помещать данные в его *LiveData*, а другой – подпишется и будет получать эти данные. Таким образом, два фрагмента будут обмениваться данными, ничего не зная друг о друге.

Чтобы два фрагмента могли работать с одной и той же моделью, они должны использовать общую активность. Код получения модели в фрагментах выглядит так:

```
SharedViewModel model = ViewModelProviders.of(getActivity())
                                          .get(SharedViewModel.class);
```

Для обоих фрагментов *getActivity*() вернет одну и ту же активность. Метод *ViewModelProviders.of* вернет провайдера активности. Далее методом *get*() можно получить модель. Код фрагмента будет выглядеть так:

```
public class MasterFragment extends Fragment {

    private SharedViewModel model;
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        model = ViewModelProviders.of(getActivity())
                                  .get(SharedViewModel.class);
        itemSelector.setOnClickListener(item -> {
            model.select(item);
        });
    }
}

public class DetailFragment extends Fragment {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        SharedViewModel model = ViewModelProviders.of(getActivity())
                                          .get(SharedViewModel.class);
        model.getSelected().observe(this, { item ->
                // Update the UI.
        });
    }
}
```

Фрагмент *MasterFragment* помещает данные в *LiveData*. А *DetailFragment* – подписывается и получает данные.

LiveData можно сравнить с *Flowable*. Но у *LiveData* есть одно большое преимущество – он учитывает состояние активности, то есть он не будет отсылать данные, если активность свернута и он отпишет от себя *Activity*, которая закрывается. *Flowable* этого не умеет. *Flowable*, который имеется в модели и на который подписана активность, будет удерживать ее, пока активность сама явно не отпишется (или пока *Flowable* не завершится).

11.5. Библиотека Room

Библиотека *Room* предоставляет удобную обертку для работы с базой данных SQLite.

Чтобы использовать *Room*, добавьте артефакты компонентов архитектуры в файл *build.gradle*:

```
dependencies {
        implementation "android.arch.persistence.room:runtime:1.0.0"
        annotationProcessor "android.arch.persistence.room:com-
piler:1.0.0"
...
    }
```

Room имеет три основных компонента: *Entity*, *DAO* и *Database*. *Entity* представляет таблицу в базе данных. *DAO* содержит методы, используемые для доступа к базе данных.

Рассмотрим их на примере, в котором будем создавать базу данных для хранения сведений о сотрудниках. Приложение использует базу данных *Room*, чтобы получить объекты доступа к данным, или *DAO*, связанным с этой базой данных. Затем приложение использует каждый *DAO* для получения сущностей из базы данных и сохранения любых изменений этих объектов обратно в базу данных. Приложение использует объект *Entity* для получения и установки значений, соответствующих столбцам таблицы в базе данных. Взаимосвязь между различными компонентами *Room* представлена на рис. 11.5.

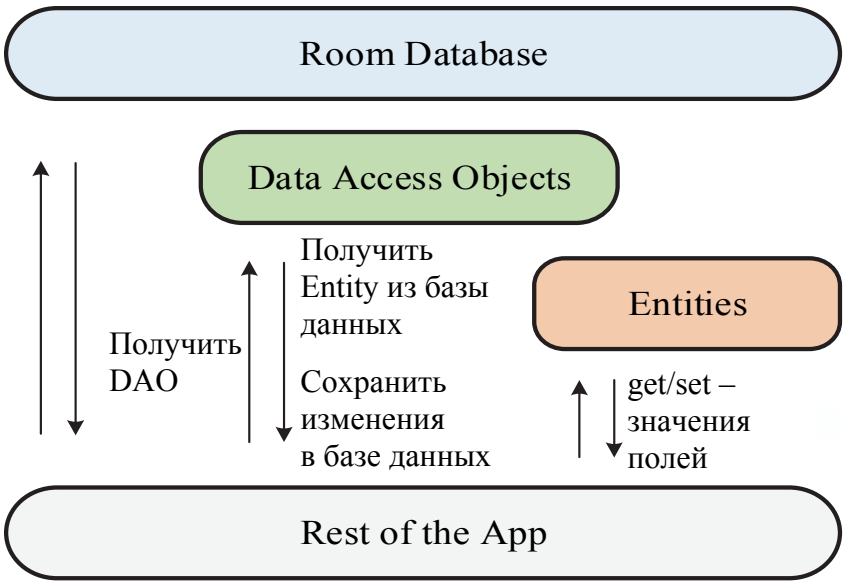


Рис. 11.5. Компоненты *Room*

Аннотацией *Entity* необходимо пометить объект, который будет храниться в базе данных:

```
@Override
@Entity
public class Employee {

    @PrimaryKey
    public long id;
    public String name;
    public int salary;
}
```

Этот же класс *Employee* будет использован для создания таблицы в базе. Аннотацией *PrimaryKey* помечаем поле, которое будет ключом в таблице.

В объекте *DAO* будут методы для работы с базой данных:

```
@Dao
public interface EmployeeDao {

    @Query("SELECT * FROM employee")
    List<Employee> getAll();

    @Query("SELECT * FROM employee WHERE id = :id")
    Employee getById(long id);

    @Insert
    void insert(Employee employee);

    @Update
    void update(Employee employee);

    @Delete
    void delete(Employee employee);

}
```

В аннотации *Query* необходимо прописать соответствующие SQL-запросы, которые будут использованы для получения данных. Для вставки/обновления/удаления используются методы *insert/update/delete* с соответствующими аннотациями. Тут никакие запросы указывать не нужно. Названия методов могут быть любыми. Главное – аннотации.

Аннотацией *Database* помечаем основной класс по работе с базой данных. Этот класс должен быть абстрактным и наследовать *RoomDatabase*.

```
@Database(entities = {Employee.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract EmployeeDao employeeDao();
}
```

В параметрах аннотации *Database* указываем, какие *Entity* будут использоваться, и версию базы. Для каждого *Entity* класса из списка *entities* будет создана таблица.

В классе *Database* необходимо описать абстрактные методы для получения *DAO*-объектов.

11.6. Использование библиотеки *Room* для работы с базой данных

Все необходимые для работы объекты созданы. *Database*-объект – стартовая точка. Чтобы его создать нужно:

```
AppDatabase db =  Room.databaseBuilder(getApplicationContext(),
                        AppDatabase.class, "database").build();
```

Необходимо использовать *Application Context*, а также указывать класс *AppDatabase* и имя файла для базы. Экземпляры *AppDatabase* очень тяжелые, поэтому рекомендуется использовать один экземпляр для всех операций. Необходимо позаботиться о синглтоне для этого объекта. Это можно сделать с помощью классов *Dagger* или *Application* для создания и хранения *AppDatabase*:

```
public class App extends Application {

    public static App instance;
    private AppDatabase database;

@Override
public void onCreate() {
  super.onCreate();
  instance = this;
  database = Room.databaseBuilder(this, AppDatabase.class, "database")
                .build();
 }

    public static App getInstance() {
        return instance;
    }

    public AppDatabase getDatabase() {
        return database;
    }
}
```

Получение базы будет выглядеть так:

```
AppDatabase db = App.getInstance().getDatabase();
```

Из *Database*-объекта получаем *DAO*:

```
EmployeeDao employeeDao = db.employeeDao();
```

Теперь можем работать с объектами *Employee*. Но эти операции должны выполняться не в UI-потоке. Иначе получим *Exception*.

Добавление нового сотрудника в базу будет выглядеть таким образом:

```
Employee employee = new Employee();
employee.id = 1;
employee.name = "John Smith";
employee.salary = 10000;
employeeDao.insert(employee);
```

Метод *getAll*() вернет всех сотрудников в *List<Employee>*:

```
List<Employee> employees = employeeDao.getAll();
```

Получение сотрудника по *id* будет выглядеть так:

```
Employee employee = employeeDao.getById(1);
```

а обновление данных по сотруднику и их удаление таким образом:

```
employee.salary = 20000;
employeeDao.update(employee);
employeeDao.delete(employee);
```

Как было сказано ранее, операции по работе с базой данных – синхронные, они должны выполняться не в UI-потоке. *Query*-операции можно сделать асинхронными, используя *LiveData* или *RxJava*. Методы *insert*/*update*/*delete* можно обернуть в асинхронный *RxJava*. Также можно использовать *allowMainThreadQueries* в *Builder* создания *AppDatabase.*

12. БАЗА ДАННЫХ И СЕРВИСЫ FIREBASE

На конференции Google I/O в 2015 г. была представлена облачная база данных на основе NoSQL с названием *Firebase*. Год спустя в мае 2016 г. на этой же конференции было объявлено об изменениях. Теперь это стала уже целая платформа для построения Android-, iOS- и мобильных веб-приложений (рис. 12.1) – https://firebase.google.com .

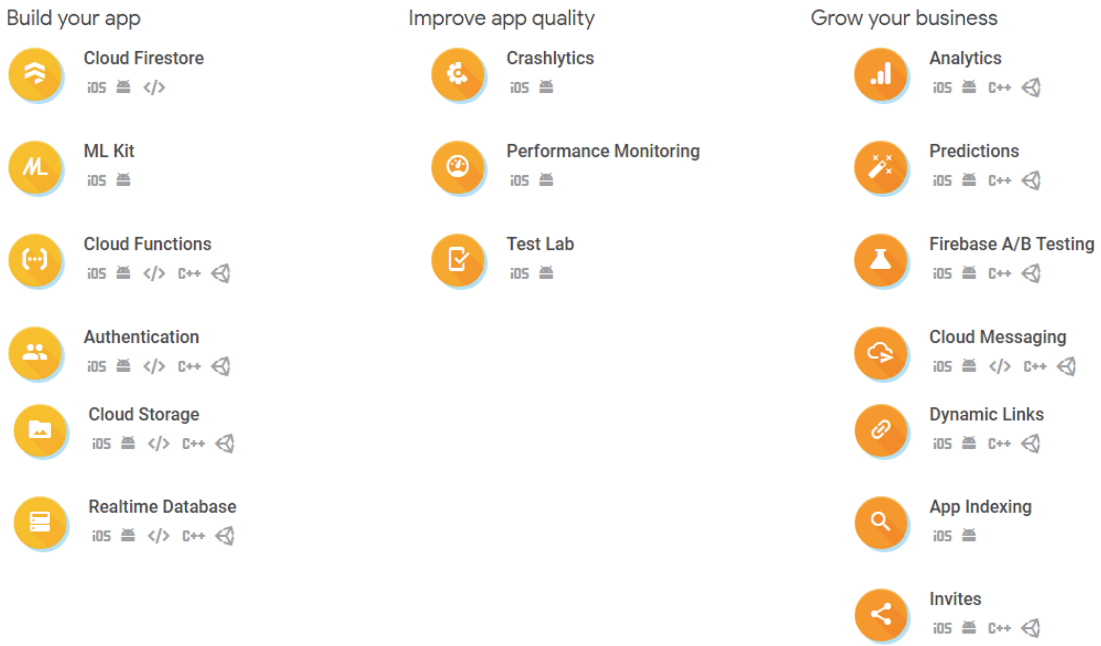


Рис. 12.1. Состав платформы *Firebase*

Firebase Realtime Database – NoSQL-база данных, которая основывается на текстовом формате обмена данными JSON.

Firebase Analytics – бесплатный инструмент для анализа приложений. Предоставляет неограниченное количество отчетов для 500 различных событий.

Firebase Cloud Messaging – бывший сервис для доставки push-уведомлений из облака.

ML Kit – мобильный SDK для машинного обучения Android и iOS. *ML Kit* предоставляет удобные API-интерфейсы, которые помогут использовать собственные модели *TensorFlow Lite* в мобильных приложениях.

Firebase Remote Config – позволяет подстраивать и обновлять элементы приложения на лету без необходимости обновлять пакет приложения. Можно включать и выключать определенные элементы приложений, распространять обновления на конкретные аудитории пользователей.

Firebase Crashlytics – репортер, который собирает важную информацию для поиска проблем iOS/Android-приложений после релиза, группирует сбои и выделяет обстоятельства, которые приводят к ним.

Firebase Test Lab – облачная инфраструктура тестирования приложений на самых разных устройствах и конфигурациях устройств. Позволяет увидеть результаты, в том числе журналы, видео и скриншоты, в консоли *Firebase*.

Firebase Dynamic Links – улучшают доступ к платформе, на которой они открывают ссылку. Если пользователь открывает динамическую ссылку на iOS или Android, их можно перенести непосредственно на связанный контент в вашем родном приложении.

Firebase Invites – готовое решение для рефералов приложений и их совместного использования через электронную почту или SMS.

Firebase Authentification – предоставляет базовые службы, простые в использовании SDK и готовые библиотеки пользовательского интерфейса для аутентификации пользователей в вашем приложении. Он поддерживает аутентификацию с использованием паролей, телефонных номеров, Google, Facebook, Twitter и др. (рис. 12.2).

Firebase App Indexing – позволяет находить пользователям результаты непосредственно из вашего приложения. Если у пользователей еще нет вашего приложения, соответствующие запросы запускают карту установки для вашего приложения в результатах поиска.

Firebase Hosting – хостинг веб-контента для разработчиков.

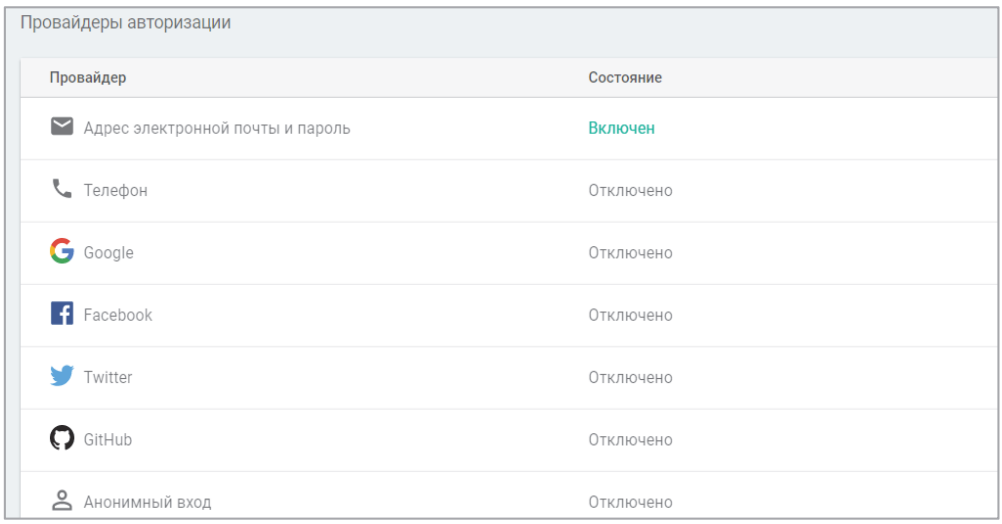


Рис. 12.2. Провайдеры аутентификации

12.1. Работа с Realtime Database

База данных позволяет работать с данными, которые хранятся как JSON, синхронизируются в реальном времени и доступны при отсутствии интернета.

В консоли Firebase (https://console.firebase.google.com/?pli=1) можно создавать новые проекты, просматривать данные пользователей, управлять файлами, работать с базой данных (рис. 12.3).

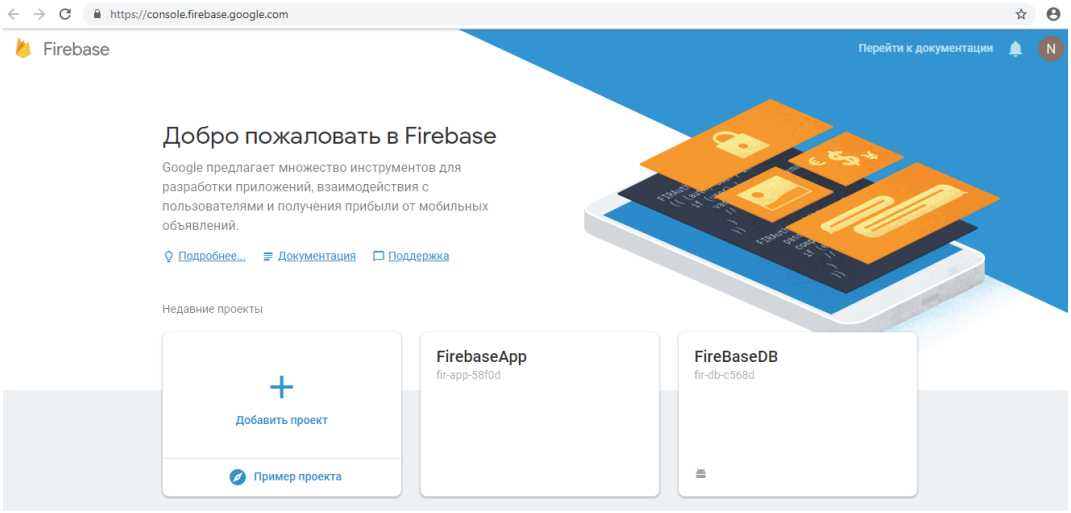


Рис. 12.3. Консоль *Firebase*

Добавление поддержки *Firebase* к проекту выполняется просто, с помощью мастера в пошаговом режиме (рис. 12.4).

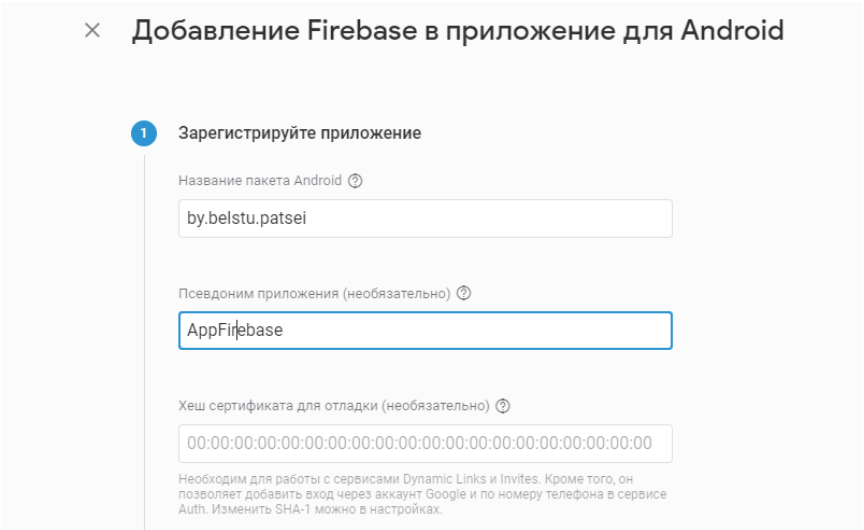


Рис. 12.4. Добавление поддержки *Firebase* к проекту. Первый шаг

Далее надо скачать файл и добавить его к проекту (рис. 12.5).

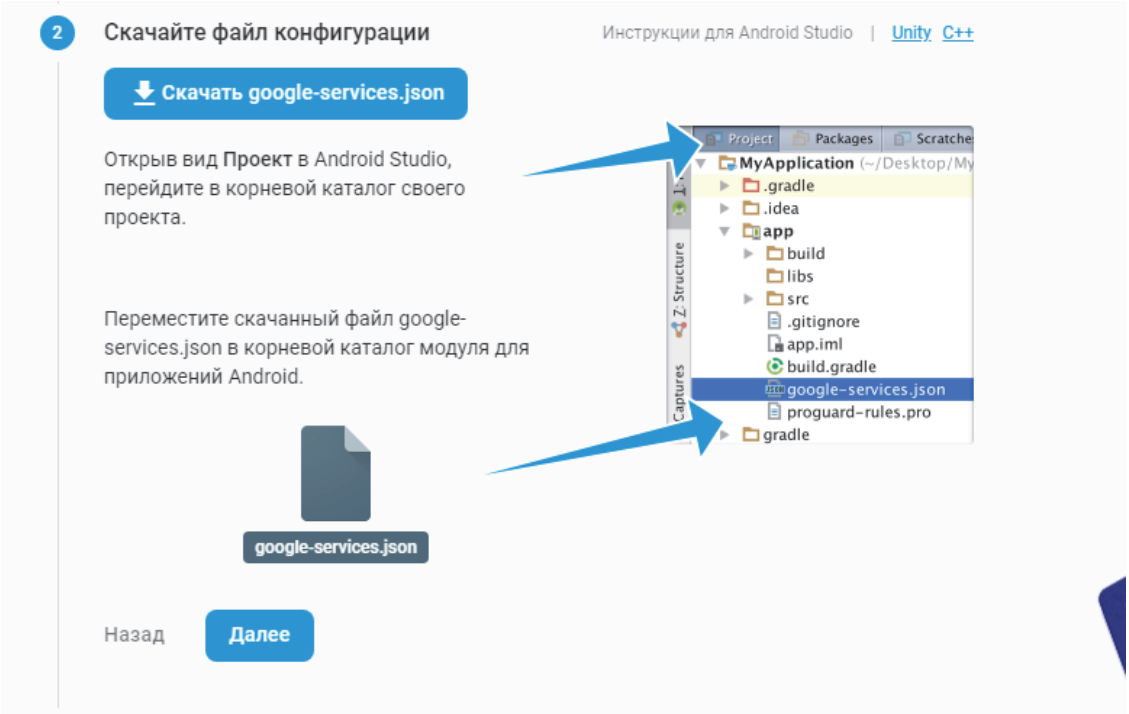


Рис. 12.5. Добавление поддержки *Firebase* к проекту. Второй шаг

Как видно, файл JSON был добавлен к проекту (рис. 12.6).

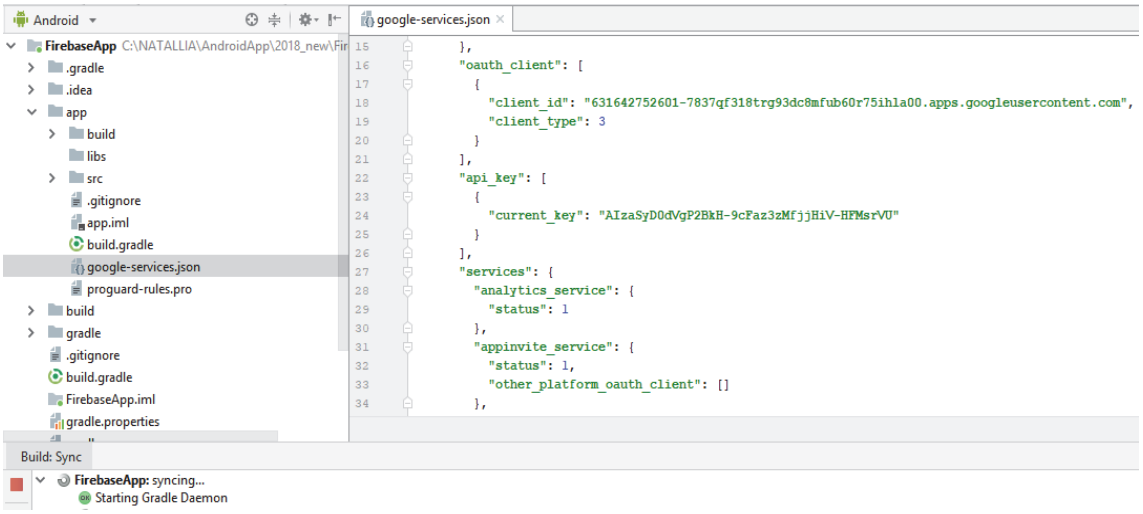


Рис. 12.6. Добавление поддержки *Firebase* к проекту. Структура файла

И третьим шагом надо добавить зависимости (рис. 12.7).

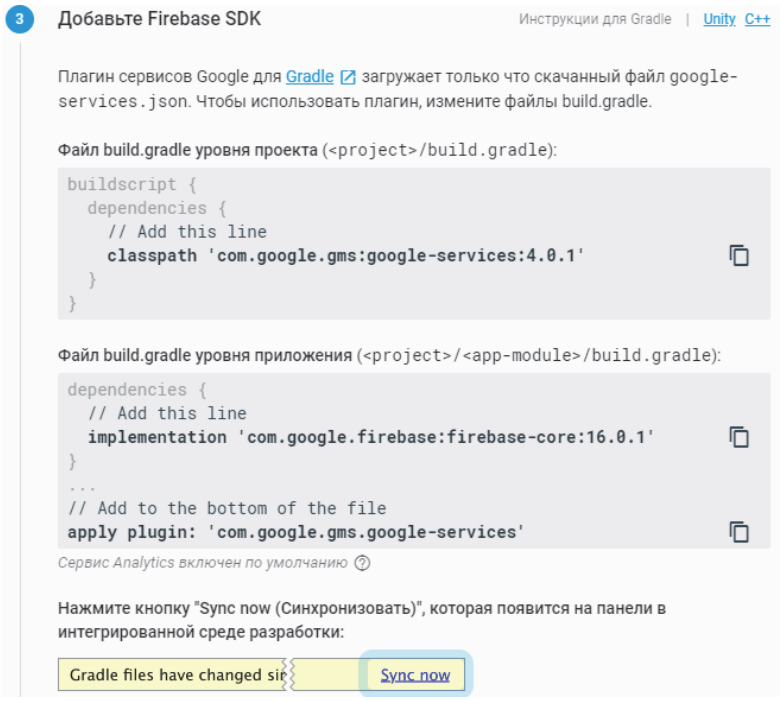


Рис. 12.7. Добавление поддержки *Firebase* к проекту. Добавление зависимости

12.2. Создание приложений с поддержкой Realtime Database

Ниже будет рассмотрен пример создания приложения с поддержкой *Realtime Database*. Для этого потребуется учетная запись *Firebase* (https://firebase.google.com), которая создается с помощью учетной записи Google.

12.2.1. Соединение с Firebase

Можно интегрировать службы *Firebase* в своем приложении прямо из Android Studio с помощью окна *Assistant*. Сначала следует убедиться, что установлен Google Repository версии 26 или выше. Теперь можно открыть и использовать окно помощника в Android Studio, выполнив следующие шаги: нажать *Tools* → *Firebase*, чтобы открыть окно *Assistant* (рис. 12.8).

12.2.2. Аутентификация

Чтобы добавить к проекту поддержку аутентификации в панели ассистента нужно выполнить 1-й и 2-й шаги (соединение и добавление зависимости) (рис. 12.9).

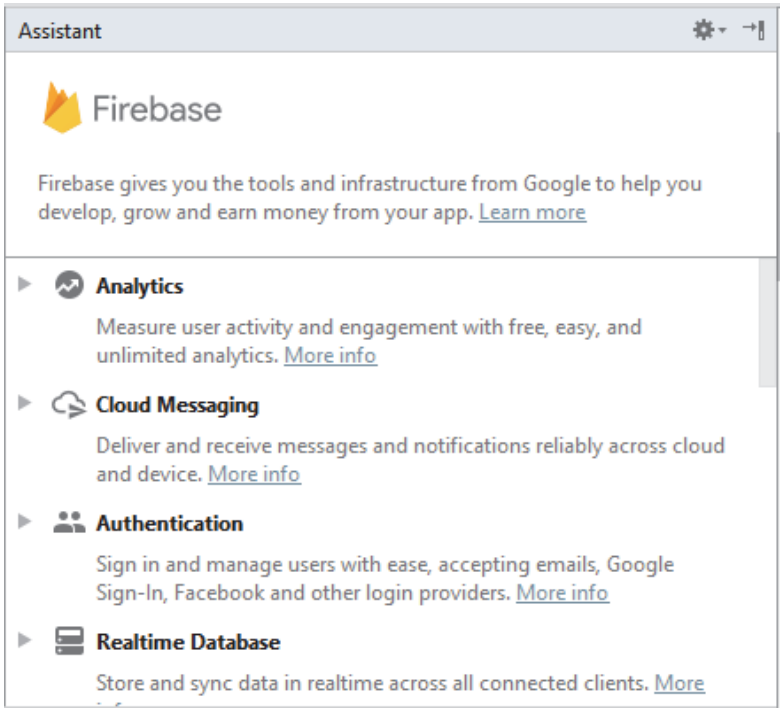


Рис. 12.8. Окно *Assistant*

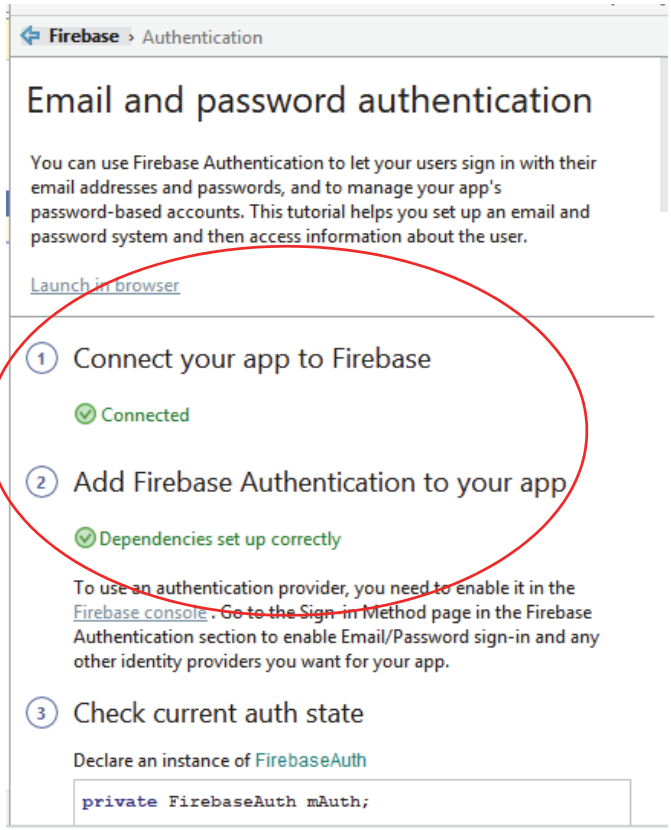


Рис. 12.9. Добавление аутентификации

Можно добавить способ входа по адресу электронной почты. Выбирается и включается провайдер авторизации (рис. 12.10).

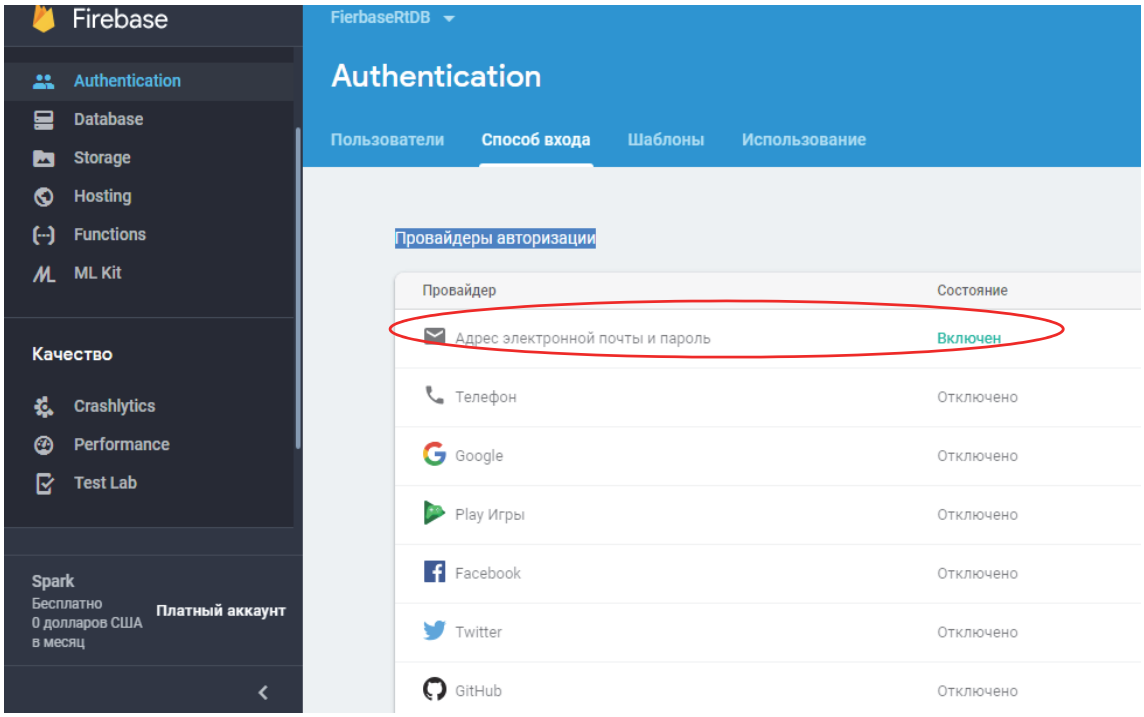


Рис. 12.10. Выбор провайдера авторизации

Создается пользователь. Для этого надо зайти на вкладку *Пользователи* и нажать кнопку *Добавить пользователя* (рис. 12.11).

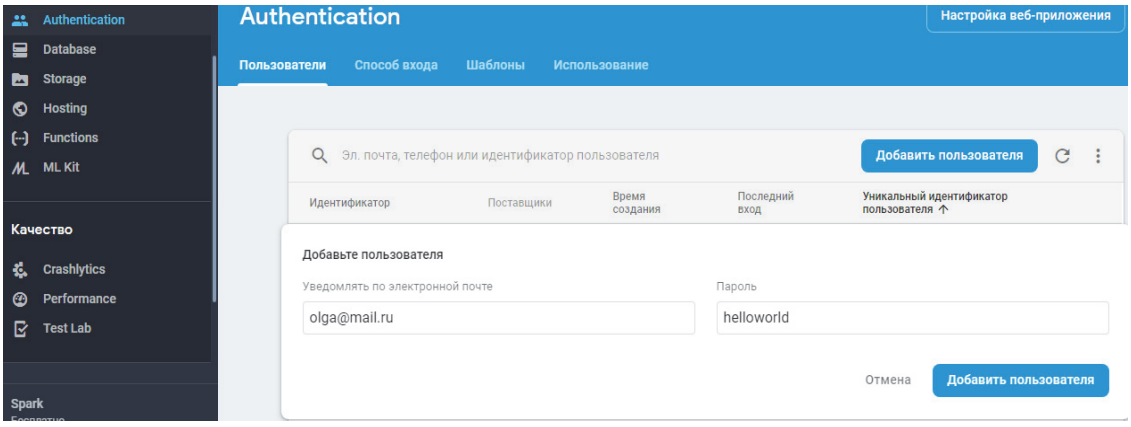


Рис. 12.11. Добавление нового пользователя

Далее выполняются шаги 3 и 4 (рис. 12.12).

Можно скопировать коды обработки входа и выхода пользователя, удаления, изменения пароля.

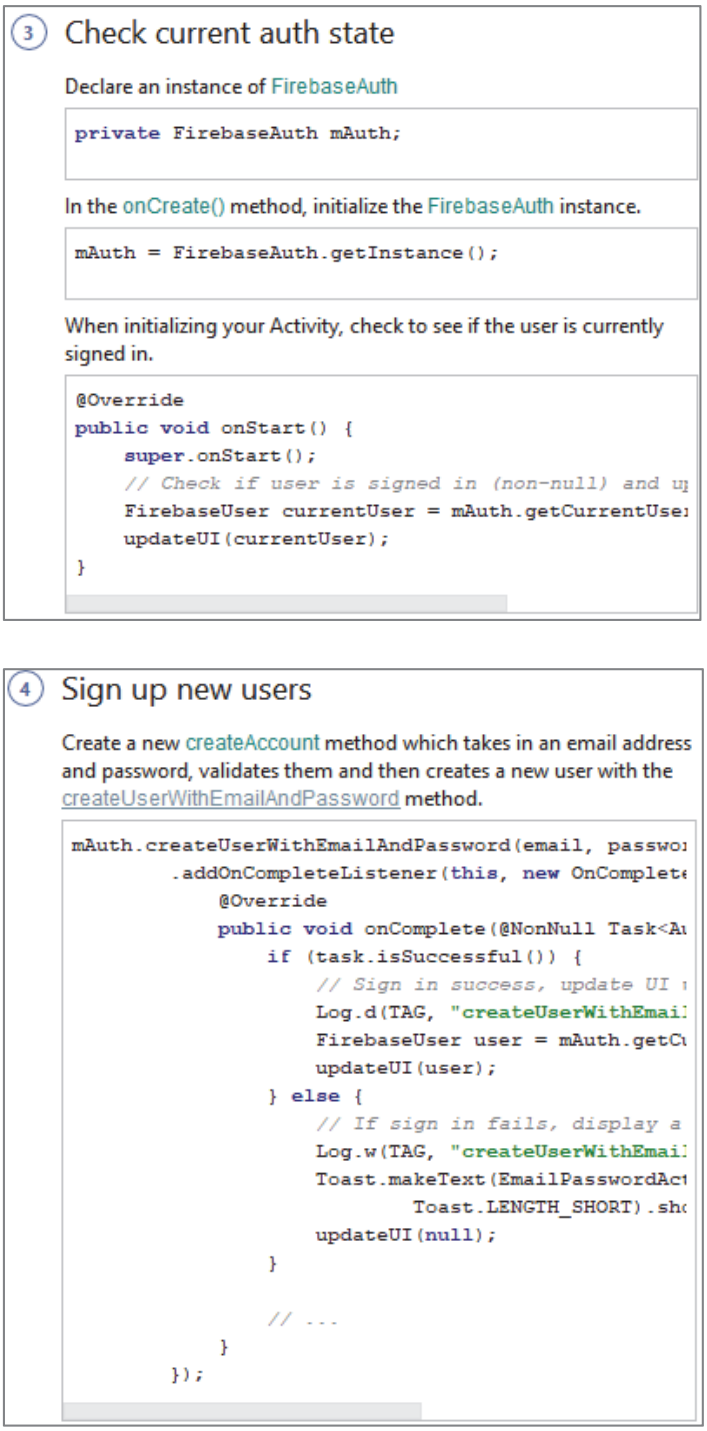


Рис. 12.12. Добавление нового пользователя

12.2.3. Добавление поддержки базы данных

Чтобы подключиться к *Firebase* и добавить необходимый код в приложение, нужно нажать на ссылку (рис. 12.13).

Затем следует выполнить шаг 1 и настроить соединение (рис. 12.14).

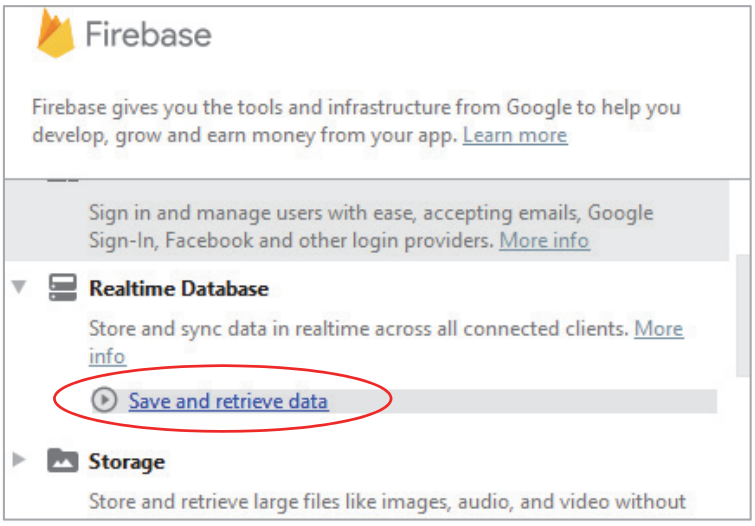


Рис. 12.13. Добавление поддержки базы данных

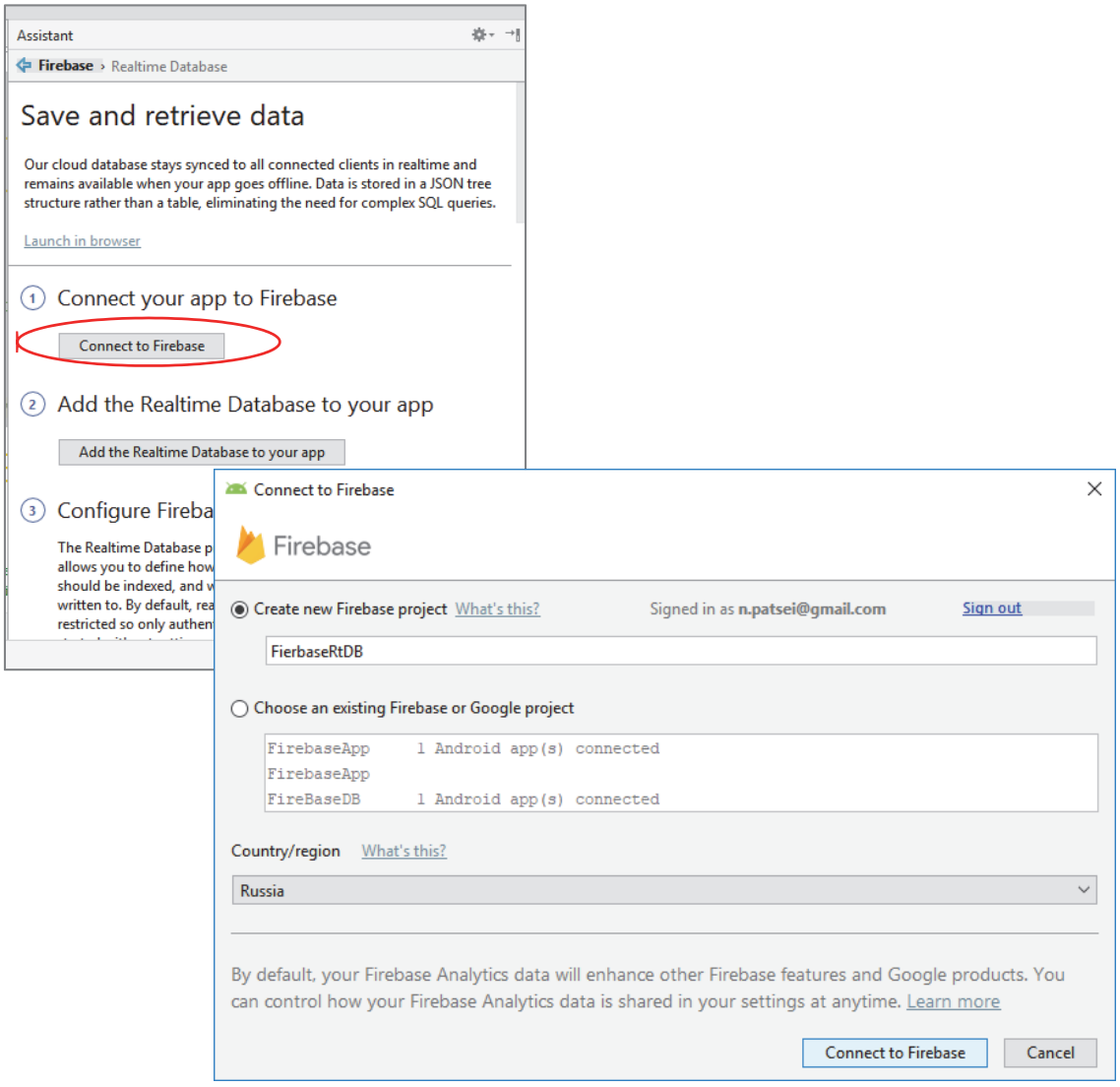


Рис. 12.14. Установление соединения с *Firebase*

Как показано на рис. 12.14, можно создать новый проект или выбрать из тех, что были созданы ранее.

Вторым шагом добавляется база данных (рис. 12.15). Разрешаются зависимости в *gradl*.

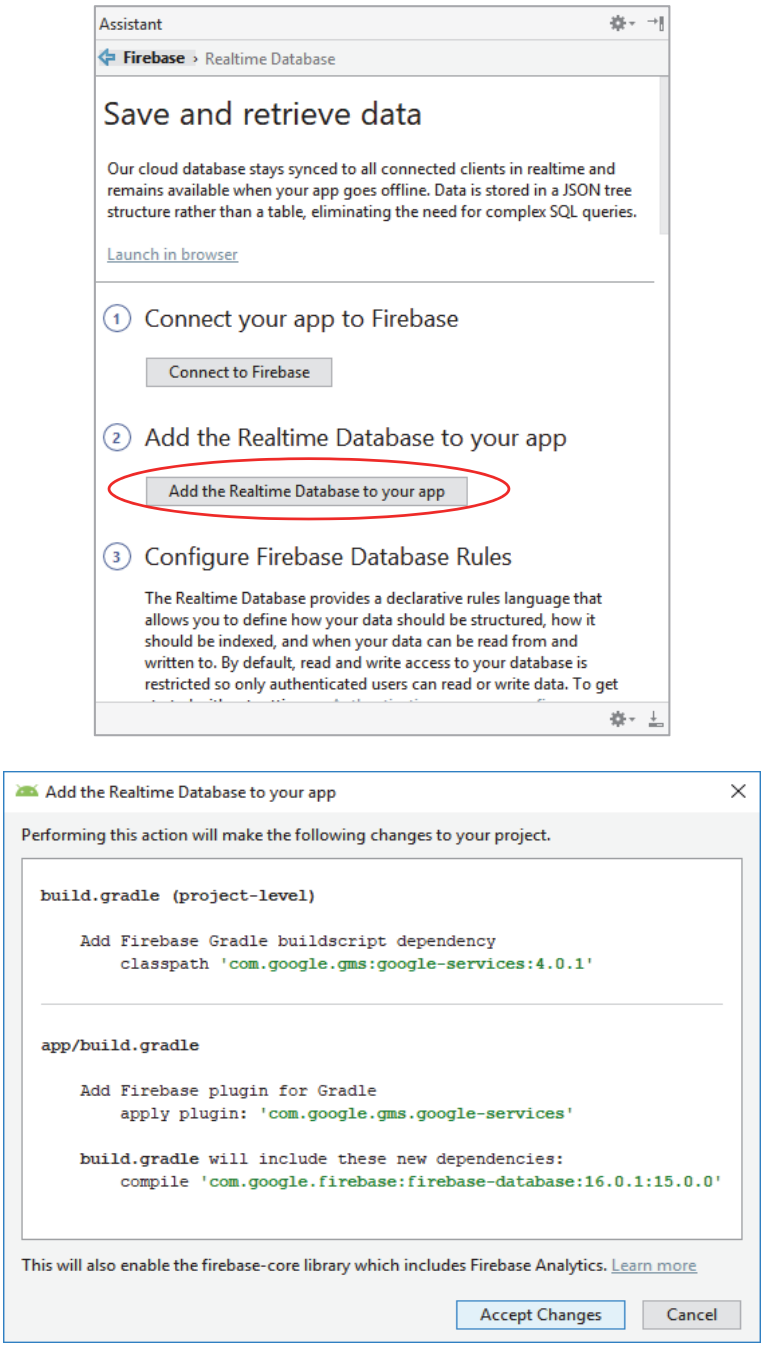


Рис. 12.15. Добавление базы данных

Третий шаг – конфигурирование правил базы данных. База данных *Firebase Realtime* предоставляет гибкий язык правил, основанных на

выражениях, подобных JavaScript, для определения структурирования, индексирования и правил чтения и записи. В сочетании со службами аутентификации можно определить, кто имеет доступ к данным и защищает личную информацию своих пользователей от несанкционированного доступа.

Это можно сделать через консоль (рис. 12.16).

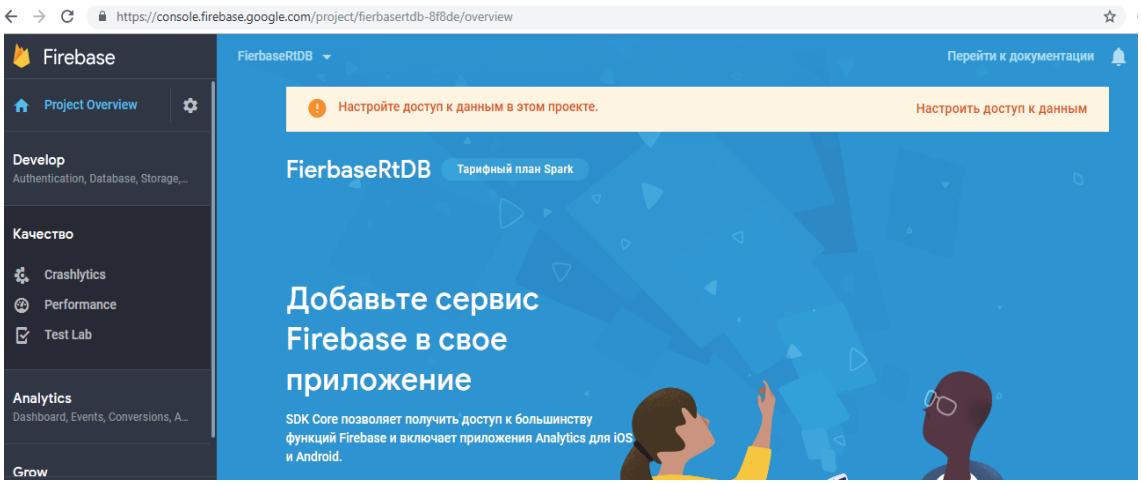


Рис. 12.16. Настройка доступа через консоль

Затем следует выбрать свой проект, нажать на раздел *Database* слева и выбрать вкладку с правилами. Чтобы протестировать правила безопасности перед их внедрением, можно имитировать операции в консоли, используя кнопку в правом верхнем углу редактора правил.

Можно также обновить правила, используя интерфейс командной строки, который позволяет программно обновлять правила, например из автоматизированной системы развертывания.

По умолчанию правила позволяют пользователю выполнять операции чтения и записи только после аутентификации (рис. 12.17).

```
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null"
  }
}
```

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

Рис. 12.17. Настройка правил работы с базой данных

Шаги 4 и 5 содержат код для чтения и записи данных (рис. 12.18).

Рис. 12.18. Запись и чтение данных

Для работы с данными создается модель *User*. Для добавления нового пользователя необходимо сделать следующее:

```
DatabaseReference mDatabase =
FirebaseDatabase.getInstance().getReference("users");
// Создаем новый узел и уникальный key value
//  /users/$userid/
    String userId = mDatabase.push().getKey();

    // Создаем
    User user = new User("Olga", "olga@mail.ru");

// Добавляем userId
mDatabase.child(userId).setValue(user);
```

Для чтения данных записывается код:

```
mDatabase.child(userId).addValueEventListener(new ValueEventListener()
{
@Override
public void onDataChange(DataSnapshot dataSnapshot) {

        User user = dataSnapshot.getValue(User.class);
        Log.d(TAG, "User name: " + user.getName() + ", email " +
                                              user.getEmail());
        }

@Override
public void onCancelled(DatabaseError error) {
        Log.w(TAG, "Failed to read value.", error.toException());
        }
});
```

13. AUDIO. VIDEO. ANIMATION

13.1. Работа с видео

Для работы с видеоматериалами в стандартном наборе виджетов Android определен класс *VideoView*, который позволяет воспроизводить видео (рис. 13.1).

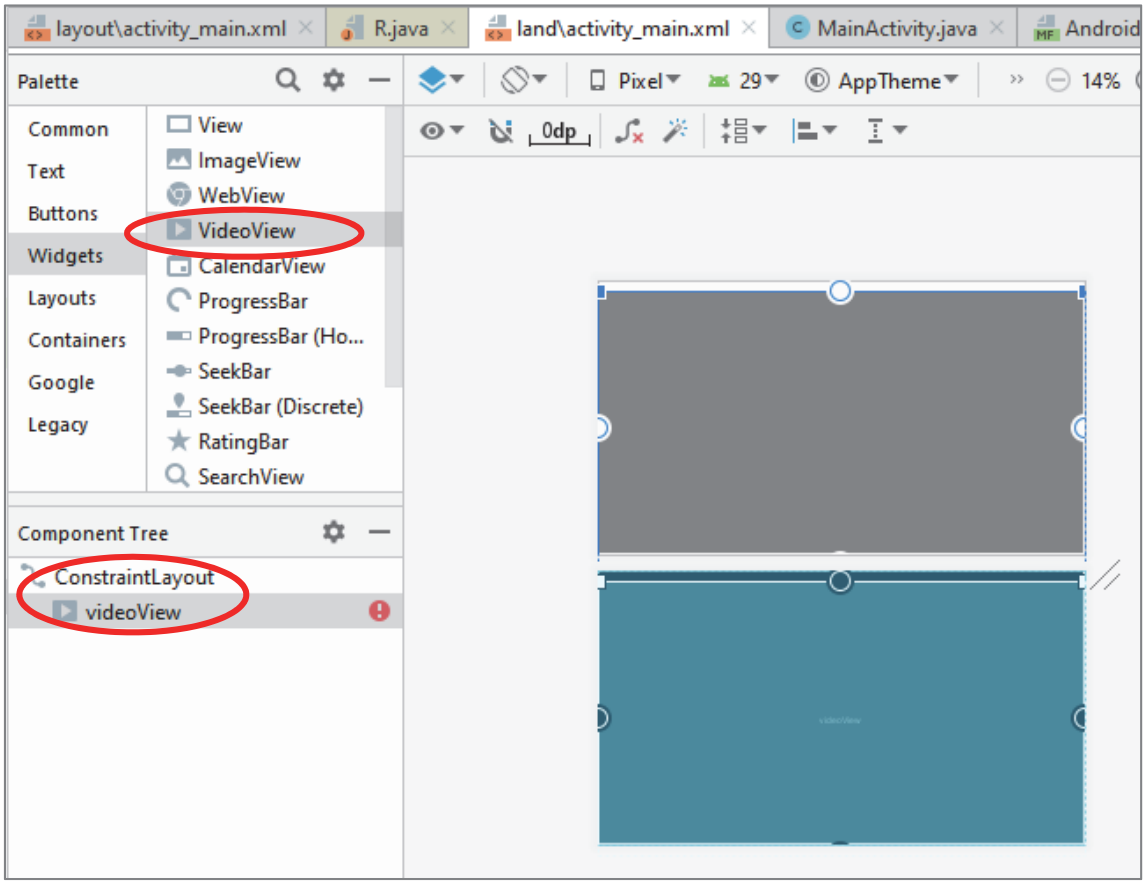


Рис. 13.1. Элементы управления для отображения видео

Android поддерживает все те же форматы видео, что и HTML 5, но также можно использовать распространенные форматы MPEG4 H.264 AVC (.mp4) и MPEG4 SP (.3gp).

VideoView может работать как с роликами, размещенными на мобильном устройстве, так и с видеоматериалами из сети (рис. 13.2).

Чтобы управлять потоком воспроизведения, надо получить объект *VideoView*. Чтобы указать источник воспроизведения, необходим объект *URI*:

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

    VideoView videoView = (VideoView) findViewById(R.id.videoView);
    videoView.setVideoPath("android.resource://" + getPackageName() +
                           "/" + R.raw.demovideo);
    MediaController mediaController = new MediaController(this);
    mediaController.setAnchorView(videoView);

    videoView.setMediaController(mediaController);
    videoView.start();
}
```

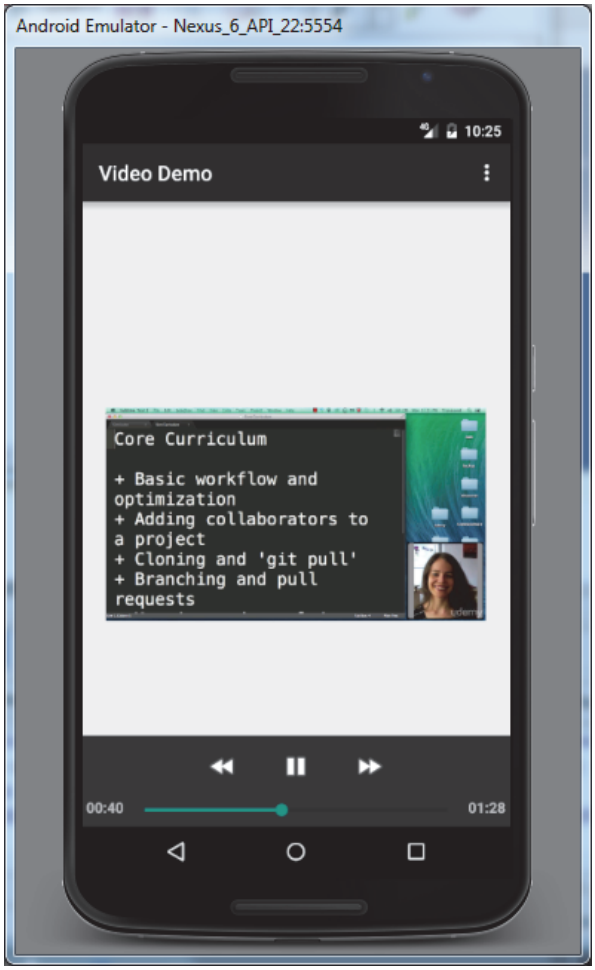


Рис. 13.2. Окно отображения *VideoView*

Метод *videoPlayer.start*() начинает или продолжает воспроизведение, метод *videoPlayer.pause*() приостанавливает видео, *videoPlayer.stopPlayback*() полностью останавливает видео. Метод *videoPlayer.resume*() позволяет снова начать воспроизведение видео сначала после его полной остановки. С помощью класса *MediaController* можно добавить к *VideoView* дополнительно элементы управления.

VideoView поддерживает воспроизведение файла из интернета. Но чтобы это стало возможно, необходимо в файле *AndroidManifest.xml* установить разрешение *android.permission.INTERNET*.

13.2. Управление аудио

Для воспроизведения музыки и других аудиоматериалов Android предоставляет класс *MediaPlayer*.

Установить нужный ресурс для воспроизведения можно тремя способами:

— в метод *create*() объекта *MediaPlayer* передается *id* ресурса, представляющего аудиофайл;

— в метод *create*() объекта *MediaPlayer* передается объект URI, представляющий аудиофайл;

— в метод *setDataSource*() объекта *MediaPlayer* передается полный путь к аудиофайлу.

После установки ресурса вызывается метод *prepare*() или *prepareAsync*() (асинхронный вариант *prepare*()). Этот метод подготавливает аудиофайл к воспроизведению, извлекая из него первые секунды. Если воспроизводить файл из сети, то лучше использовать *prepareAsync*().

```
MediaPlayer mediaPlayer =  new MediaPlayer().create(this, R.raw.somg);

mediaPlayer.start();
mediaPlayer.pause();
mediaPlayer.stop();
```

Для управления громкостью звука применяется класс *AudioManager*:

```
AudioManager audio = (AudioManager)
                        getSystemService(Context.AUDIO_SERVICE);
int maxVol = audio.getStreamMaxVolume(AudioManager.STREAM_MUSIC);
int curVol= audio.getStreamVolume(AudioManager.STREAM_MUSIC);
audio.setStreamVolume(AudioManager.STREAM_MUSIC,curVol,0);
```

Для записи можно использовать *MediaRecorder*. Чтобы *MediaRecorder* записал звук, он должен знать:

– источник звука;
– формат записи;
– аудиокодек;
– имя файла.

AudioRecorder не пишет данные, а позволяет их получать в приложении, то есть является посредником между приложением и микрофоном. *AudioRecorder* начинает получать данные с микрофона и хранит их у себя во внутреннем буфере. Можно при создании *AudioRecorder* указать желаемый размер этого буфера и далее запрашивать из него данные методом *read*().

AudioRecorder будет полезен, если нужно как-то обработать данные перед записью в файл или отправить данные не в файл, а куда-либо еще.

13.3. Анимация

Анимации могут добавлять визуальные подсказки, которые уведомляют пользователей о том, что происходит в вашем приложении. Они полезны, когда пользовательский интерфейс изменяет состояние, например, когда загружается новый контент или становятся доступными новые действия. Анимации придают приложению более качественный вид.

Android включает в себя различные API-интерфейсы анимации в зависимости от того, какой тип анимации будет использоваться.

По объекту применения можно классифицировать анимации на следующие виды.

Анимированные bitmaps. Если нужно анимировать растровое изображение, такое как значок или иллюстрация, можно использовать *Drawable Animation* API. Обычно эти анимации определяются статически с помощью ресурса, но можно определить поведение анимации во время выполнения.

Анимация UI-видимости и движения. Когда нужно изменить видимость или положение представлений в макете, следует включить UI-анимации, чтобы помочь пользователю понять, как меняется пользовательский интерфейс.

Для перемещения, раскрытия или скрытия представлений в текущем макете можно использовать ***анимацию свойств***, предоставляемую

пакетом android.animation, доступным с Android 3.0 (уровень API 11) и выше. Эти API-интерфейсы обновляют свойства объектов *View* в течение определенного периода времени, непрерывно перерисовывая представление по мере изменения свойств. Для создания таких анимаций с наименьшими усилиями можно включить ее в макет, чтобы при изменении видимости представления анимация применялась автоматически.

Physics-based motion. В любом возможном случае анимации должны применять физику реального мира, чтобы выглядеть естественно. Например, они должны поддерживать импульс, когда их цель изменяется, и делать плавные переходы во время любых изменений.

Есть две распространенные анимации на основе физики: *Spring Animation* и *Fling Animation*.

Анимации, не основанные на физике, например созданные с помощью *API ObjectAnimator*, довольно статичны и имеют фиксированную продолжительность.

Анимация изменений Layout. На Android 4.4 (уровень API 19) и выше можно использовать инфраструктуру перехода для создания анимации при смене макета внутри активности или фрагмента. Для этого следует указать начальный и конечный макеты, а также тип анимации, которая будет использоваться. Затем система вычисляет и выполняет анимацию между двумя макетами. Начальный и конечный макеты хранятся в *Scene*, хотя начальная сцена обычно определяется автоматически из текущего макета. Далее создается переход для сообщения системе типа анимации и вызывается *TransitionManager.go*(). Система запускает анимацию, чтобы поменять местами макеты.

Анимация между Activity. На Android 5.0 (уровень API 21) и выше можно создавать анимации, которые переходят между активностями. Это основано на той же структуре переходов, которая описана выше для анимации изменений макета, но она позволяет создавать анимации между макетами в отдельных активностях.

13.3.1. View Animation

Эта анимация (поддерживается всеми версиями Android SDK) используется для любых объектов, расширяющих *View* путем изменения свойств: position (положения), size (размер), rotation (поворот), transparency (прозрачность). Преобразования во времени вычисляются автоматически. Трансформации конфигурируются в XML-файлах, затем считываются и присваиваются *View*-элементам в коде программы.

Создание анимации (рис. 13.3). В проекте есть папка *res*. В ней надо создать папку *anim: res* → *anim* и затем файл *someanimation.xml*.

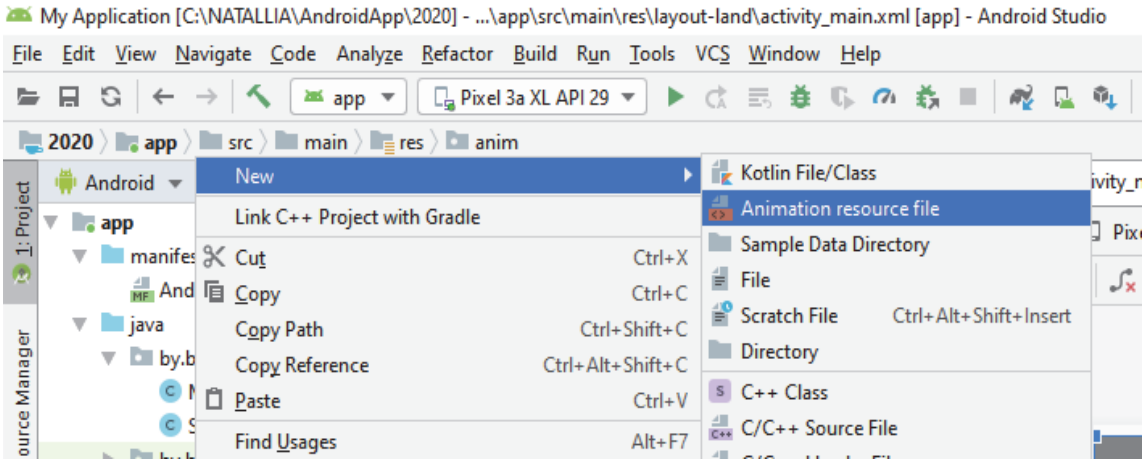


Рис. 13.3. Создание файла анимации

Определить анимацию для прозрачности, масштабировния, поворота, перемещения:

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

    <scale android:fromXScale="1.0"
        android:fromYScale="1.0"
        android:toXScale="0.5"
        android:toYScale="0.5"
        />
    <alpha android:fromAlpha="0"
        android:toAlpha="1"/>
</set>
```

Можно использовать теги *<alpha>*, *<scale>*, *<translate>*, *<rotate>* и *<set>*, которые унаследованны от базового класса *Animation* и поэтому имеют атрибуты:

— *duration* – продолжительность эффекта;

— *startOffset* – начальное время смещения;

— *fillBefore* – когда установлен в *true*, то преобразование анимации применяется перед началом анимации;

— *fillAfter* – когда установлен в *true*, то преобразование применяется после конца анимации;

— *repeatCount* – число повторений анимации;

— *repeatMode* – поведение анимации при ее окончании: restart (перезапустить без изменений) или reverse (изменить анимацию в обратном направлении);

— *zAdjustment* – режим упорядочения оси *z*;

— *interpolator* – скорость изменения анимации.

Затем следует загрузить файл анимации и запустить ее:

```
public class MainActivity extends ActionBarActivity {
    private ImageView drag;
    public void onImageClick(View view) {

        Animation animation =  AnimationUtils
                  .loadAnimation(this,R.anim.someanimation);
        drag.startAnimation(animation);

    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
         drag = (ImageView) findViewById(R.id.dragon);
    }
```

Тегам XML соответствуют классы:

— *AnimationSet* (элемент <*set*>) – класс, представляющий группу анимаций, которые должны запускаться вместе;

— *AlphaAnimation* (элемент <*alpha*>) – управляет прозрачностью объекта;

— *RotateAnimation* (элемент <*rotate*>) – регулирует вращение объекта;

— *ScaleAnimation* (элемент <*scale*>) – управляет масштабированием объекта, то есть изменениями размеров;

— *TranslateAnimation* (элемент <*translate*>) – управляет позиционированием объекта (перемещением).

Для любого *View*-объекта с API 12 и выше можно применить метод *animate*():

```
drag.animate()
        .scaleY(2)
        .scaleY(2)
        .rotationX(180)
        .rotationY(180)
        .translationX(200)
        .translationY(200)
        .setDuration(2000);
```

13.3.2. Frame (Drawable) Animation

Drawable Animation позволяет загрузить последовательность *Drawable*-ресурсов один за другим, чтобы в результате получилась анимация. Это традиционный способ анимации, поддерживаемый всеми устройствами. Создается воспроизведением разных картинок, проигрываемых в определенном порядке, наподобие фильма. Класс *AnimationDrawable* является базовым для анимации такого типа.

В файле XML перечисляются фреймы, составляющие анимацию (рис. 13.4).

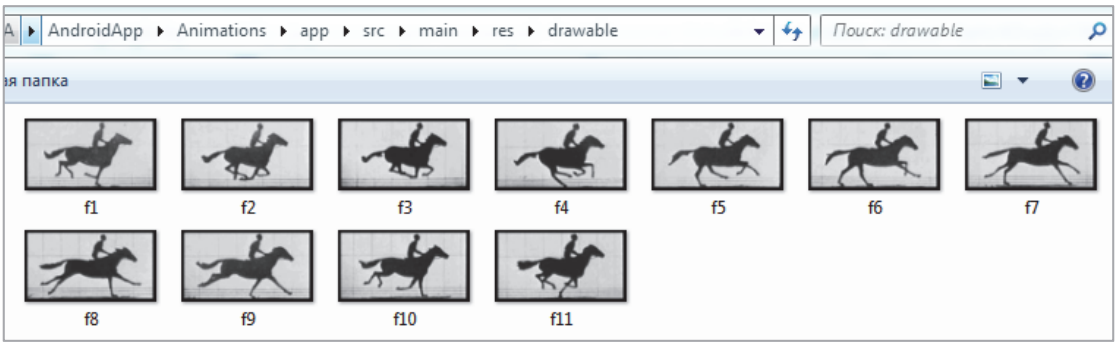


Рис. 13.4. Фреймы для анимации

Файл должен принадлежать директории *res/drawable/* проекта. В этом случае в файле XML могут быть заданы инструкции по порядку воспроизведения и длительности для каждого фрейма в анимации.

Файл XML состоит из элемента <*animation-list*> в качестве корневого и последовательных дочерних узлов <*item*>, которые задают каждый фрейм: *drawable*-ресурс для фрейма и длительность фрейма. Вот пример файла XML для *drawable*-анимации:

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.com/apk/res/an-
droid">
<item android:drawable="@drawable/f1" android:duration="50"/>
    <item android:drawable="@drawable/f1" android:duration="50"/>
    <item android:drawable="@drawable/f2" android:duration="50"/>
    <item android:drawable="@drawable/f3" android:duration="50"/>
    <item android:drawable="@drawable/f4" android:duration="50"/>
    <item android:drawable="@drawable/f5" android:duration="50"/>
    <item android:drawable="@drawable/f6" android:duration="50"/>
    <item android:drawable="@drawable/f7" android:duration="50"/>
    <item android:drawable="@drawable/f8" android:duration="50"/>
    <item android:drawable="@drawable/f9" android:duration="50"/>
</animation-list>
```

После этого создается *layout*:

```
<ImageView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/house"
        android:onClick="onImageClick" />
</RelativeLayout>
```

и запускается анимация:

```
import android.graphics.drawable.AnimationDrawable;
...
public class MainActivity extends ActionBarActivity {

    private  ImageView house;
    private AnimationDrawable houseAnimation;

@Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
      house = (ImageView) findViewById(R.id.house);
        if (house==null) throw new AssertionError();

      house.setBackgroundResource(R.drawable.house_animation);

        houseAnimation = (AnimationDrawable)    house.getBackground();
        houseAnimation.start();
    }

public void onImageClick(View view) {
        if (houseAnimation.isRunning())
            houseAnimation.stop();
        else
            houseAnimation.start();
    }
```

Метод *start*() экземпляра *AnimationDrawable* не может быть вызван во время метода *onCreate*() активности, потому что *AnimationDrawable* еще не полностью подключен к окну приложения. Если надо проиграть анимацию немедленно, без необходимости взаимодействия с пользователем, то следует вызвать ее из метода жизненного цикла, когда окно приложения Android получает фокус.

13.3.3. Property Animation

Property Animation поддерживается с Android 3.0 API 11. Может быть применена к любому объекту и свойству. Его основным инструментом является *Animator*.

Animator – это тип, предназначенный для изменения значений выбранного объекта относительно времени. Это инструмент для управления потоком заданной длительности, который изменяет определенное свойство от начального значения к конечному.

Идеологическое отличие классов *Animator* от *View Animation* в том, что *View Animation* изменяет «представление» объекта, не изменяя сам объект.

ValueAnimator наследуется от *Animator*. В самом простом варианте следует задать этому классу тип изменяемого значения, начальное

и конечное значения и запустить. В ответ будут приходить события на начало, конец, повторение и отмену анимации и еще на два события, которые задаются отдельно для паузы и изменения значения. Событие изменения самое важное: в него будет приходить измененное значение, с помощью которого меняются свойства объектов.

Ниже рассматривается пример расчета и вывода значения для *value*. Используя слушателя выведем промежуточные значения на экран в лог (результат на рис. 13.5):

```
ValueAnimator  valueAnimator = new ValueAnimator()
                               .ofFloat(1f,10f).setDuration(100);
    valueAnimator.addUpdateListener(
          new ValueAnimator.AnimatorUpdateListener() {
        @Override
        public void onAnimationUpdate(ValueAnimator animation) {
            Log.d("time",
                  String.valueOf(animation.getCurrentPlayTime()));
            Log.d("value",
                  String.valueOf(animation.getAnimatedValue()));
        }
    }); valueAnimator.start();
}
```

```
:k: AUDIO_OUTPUT_FLAG_FAST deni
s D/time: 2
s D/value: 1.0
s D/time: 18
s D/value: 1.6266606
s D/time: 35
s D/value: 3.332109
s D/time: 52
s D/value: 5.5
s D/time: 68
s D/value: 7.790687
s D/time: 84
s D/value: 9.443379
s D/time: 101
s D/value: 10.0
```

Рис. 13.5. Изменения значения
на основе *ValueAnimator*

ObjectAnimator наследуется от *ValueAnimator*. Класс предназначен упростить работу с *ValueAnimator*. Необходимо задать классу *Animator* объект и указать поле, которое надо изменить, например *scaleX*. Затем надо найти *setter* для этого поля (в данном случае – *setScaleX*). Далее *Animator* самостоятельно будет менять значение этого поля.

Пример:

```
public class MainActivity extends ActionBarActivity {

        private ImageView drag;

 @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
            drag = (ImageView) findViewById(R.id.dragon);
    }

    public void onImageClick(View view) {
      ObjectAnimator objectAnimator = new ObjectAnimator()
                    .ofFloat(drag,"scaleX",1f,0.5f).setDuration(2000);
      objectAnimator.start();
    }
```

Существует ряд полезных классов для работы с анимацией свойств. Например, *AnimatorSet* (наследуется от *Animator*) – механизм объединения анимаций вместе, хореография по отношению друг к другу:

```
ObjectAnimator objectAnimator = new ObjectAnimator()
          .ofFloat(drag,"scaleX",1f,0.5f).setDuration(2000);
//  objectAnimator.start();
  ObjectAnimator objectAnimator2 = new ObjectAnimator()
          .ofFloat(drag,"scaleY",1f,0.5f).setDuration(2000);
 // objectAnimator2.start();

  AnimatorSet animationSet = new AnimatorSet();
  animationSet.playTogether(objectAnimator,objectAnimator2);
  animationSet.start();
```

ViewPropertyAnimator – это отдельный класс, не наследуется от *Animator*, но обладает той же логикой, что и *ObjectAnimator* для *View*, и предназначен для легкого анимирования *View*.

Evaluator (*оценки*) – сообщают системе анимации свойств, как вычислять значения для данного свойства.

Interpolator – класс, который модифицирует анимацию с использованием математических вычислений (включенные в SDK и собственные) и определяет, как рассчитываются значения (функция времени):

— *AccelerateDecelerateInterpolator* – интерполятор, скорость изменения которого начинается и заканчивается медленно, но ускоряется в середине;

— *AccelerateInterpolator* – интерполятор, скорость изменения которого начинается медленно, а затем ускоряется;

— *AnticipateInterpolator* – интерполятор, чье изменение начинается с конца, затем движется вперед;

— *AnticipateOvershootInterpolator* – интерполятор, чье изменение начинается с конца, движется вперед и перевыполняет целевое значение, затем возвращается к значению;

— *BounceInterpolator* – интерполятор, чье изменение отскакивает в конце;

— *CycleInterpolator* – интерполятор, анимация которого повторяется для определенного количества циклов;

— *DecelerateInterpolator* – интерполятор, скорость изменения которого начинается быстро и затем замедляется;

— *LinearInterpolator* – интерполятор, скорость изменения которого постоянна;

— *OvershootInterpolator* – интерполятор, чье изменение движется вперед, перевыполняет конечное значение, затем возвращается;

— *TimeInterpolator* – интерфейс для собственного интерполятора.

Keyframes – технология, которая состоит из пары «время – значение», которая позволяет определить состояние в определенное время анимации. Каждый ключевой кадр может также иметь свой собственный интерполятор:

```
Keyframe kf0 = Keyframe.ofFloat(0f, 0f);
Keyframe kf1 = Keyframe.ofFloat(.5f, 360f);
Keyframe kf2 = Keyframe.ofFloat(1f, 0f);
PropertyValuesHolder pvhRotation =
      PropertyValuesHolder.ofKeyframe("rotation", kf0, kf1, kf2);
ObjectAnimator rotationAnim =
         ObjectAnimator.ofPropertyValuesHolder(target, pvhRotation)
rotationAnim.setDuration(5000);
```

Объявление анимаций можно задавать в XML. Для *ValueAnimator* используют тег *<animator>*, для *ObjectAnimator* – *<objectAnimator>*, *AnimatorSet* – *<set>*.

```
<set android:ordering="sequentially">

    <set>
        <objectAnimator
            android:propertyName="x"
            android:duration="500"
            android:valueTo="400"
            android:valueType="intType"/>
        <objectAnimator
            android:propertyName="y"
            android:duration="500"
            android:valueTo="300"
            android:valueType="intType"/>
    </set>
    <objectAnimator
        android:propertyName="alpha"
        android:duration="500"
        android:valueTo="1f"/>

</set>
```

13.3.4. Transitions Framework

Начиная с Android 4.4 (API 19) появился дополнительный инструмент для создания анимаций – *Transitions Framework*. Он позволяет анимировать все виды движения в пользовательском интерфейсе, просто предоставляя начальный макет и конечный. Можете выбрать тип анимации (например, для затухания или уменьшения размеров) (рис. 13.6).

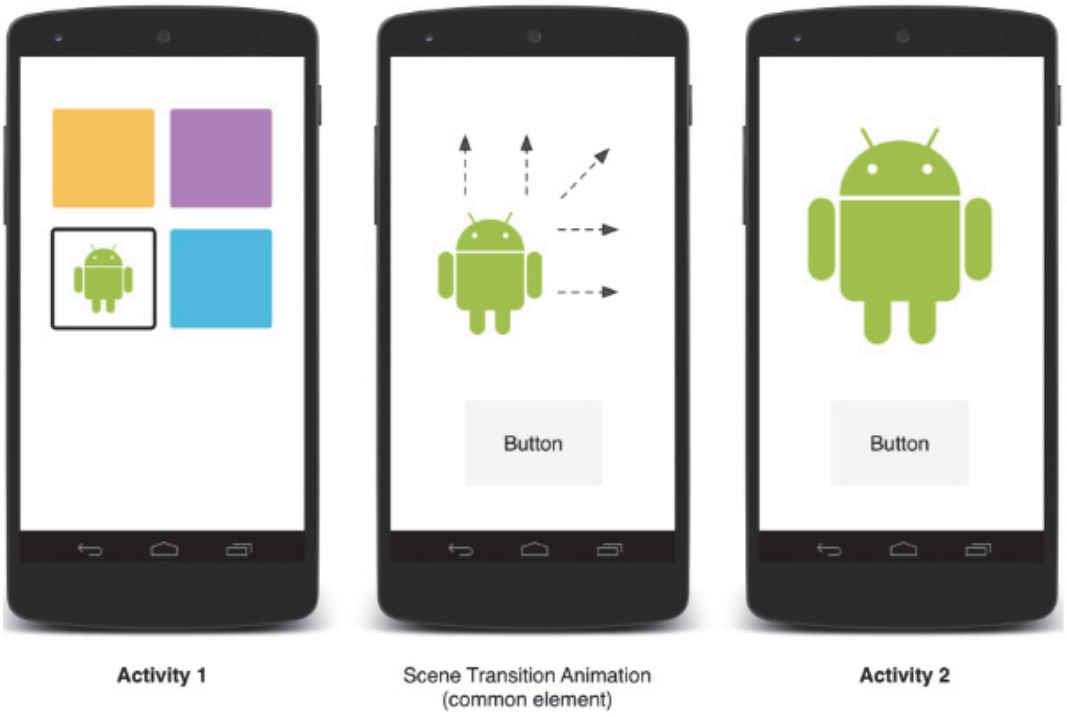


Рис. 13.6. *Transition Animation*

Transition Framework включает в себя следующие функции:

— *group-level animations* – применяет один или несколько эффектов анимации ко всем представлениям в иерархии представлений;

— *built-in animations* – использует предопределенные анимации для общих эффектов, таких как затухание или движение;

— *поддержка файлов ресурсов* – загрузка иерархий представления и встроенных анимаций из файлов ресурсов макета;

— *обратные вызовы жизненного цикла* – получение обратных вызовов, которые обеспечивают контроль над процессом анимации и изменения иерархии.

Основной процесс анимации между двумя макетами выглядит следующим образом. Создается объект *Scene* для начального и конечного макетов. Сцена начального макета часто определяется автоматически

из текущего. Создается объект *Transition*. Вызывается *TransitionManager.go*(), и система запускает анимацию, чтобы поменять местами макеты (рис. 13.7).

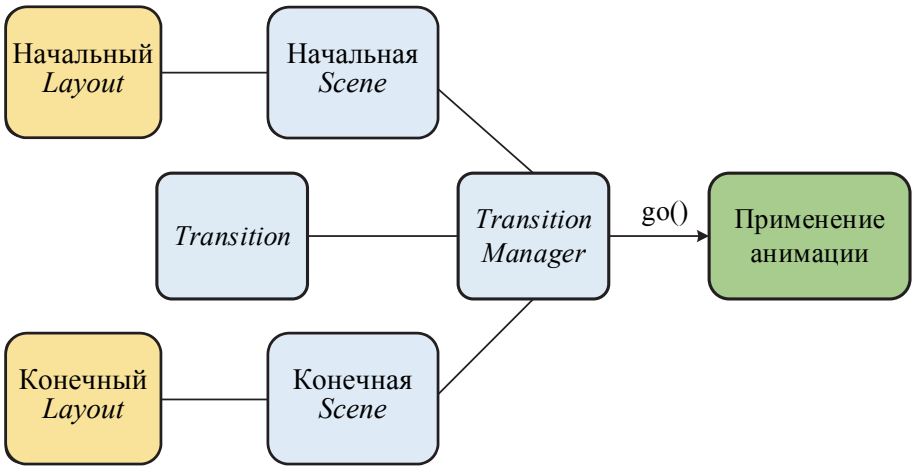


Рис. 13.7. Схема работы *Transitions Framework*

Scene – последовательность *View*, организованных иерархически (XML *Layout*). *Transition* – определяет аниматор переходов, который показывает, как должны сменяться сцены (определяется в XML). *TransitionInflater* – класс, который читает изменения XML. *TransitionManager* – управляет изменениями во время выполнения.

У этого типа анимации существуют ограничения. Так, анимации, примененные к *SurfaceView*, могут отображаться неправильно. Экземпляры *SurfaceView* обновляются из потока, не являющегося пользовательским интерфейсом, поэтому обновления могут быть не синхронизированы с анимацией других представлений. Некоторые конкретные типы переходов могут не давать желаемого эффекта анимации при применении к *TextureView*. Классы, расширяющие *AdapterView*, такие как *ListView*, управляют своими дочерними представлениями с помощью способов, несовместимых со структурой переходов.

Ниже рассмотрен пример создания анимации между двумя сценами, показанными на рис. 13.8. Сначала создается макет:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/an-
droid"
    android:id="@+id/master_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <FrameLayout
        android:id="@+id/scene_root"
```

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

        <include layout="@layout/scene1" />
    </FrameLayout>
</LinearLayout>
```

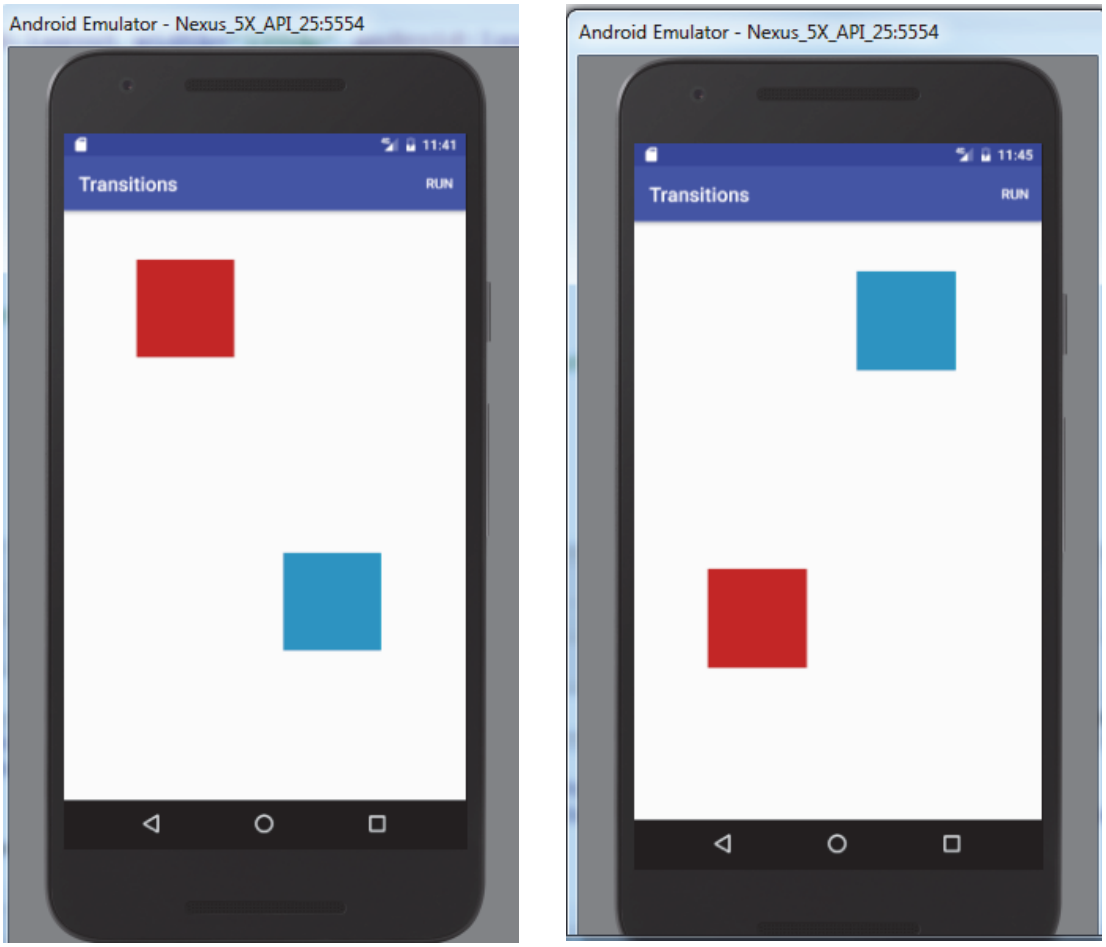


Рис. 13.8. Сцены для анимации

Затем создается *scene1.xml*:

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

    <ImageView
        android:id="@+id/imageViewRed"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:layout_marginStart="75dp"
        android:layout_marginTop="50dp"
        android:background="@android:color/holo_red_dark" />
   <ImageView
```

```
        android:id="@+id/imageViewBlue"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:layout_marginStart="225dp"
        android:layout_marginTop="350dp"
        android:background="@android:color/holo_blue_dark" />

</RelativeLayout>
```

Потом *scene2.xml:*

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

    <ImageView
        android:id="@+id/imageViewRed"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:layout_marginStart="75dp"
        android:layout_marginTop="350dp"
        android:background="@android:color/holo_red_dark" />

    <ImageView
        android:id="@+id/imageViewBlue"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:layout_marginStart="225dp"
        android:layout_marginTop="50dp"
        android:background="@android:color/holo_blue_dark" />

</RelativeLayout>
```

После того как определили начальную и конечную сцены, между которыми нужно переключиться, создается объект *Transition*, который определяет анимацию. Каркас позволяет указать встроенный переход в файле ресурсов и создать экземпляр встроенного перехода в коде.

Часто используют простые типы *Transition. ChangeBounds* – для изменения координат *View* внутри *Layout* и его размеров. *Fade* – для появления и исчезания.

Определяется *Transition* или в XML, или в Java-коде. Если делать анимацию через XML надо добавить каталог *res*/*transition*/ в проект. Создать новый файл ресурсов XML (рис. 13.9) для одного из встроенных переходов *first_transition*:

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

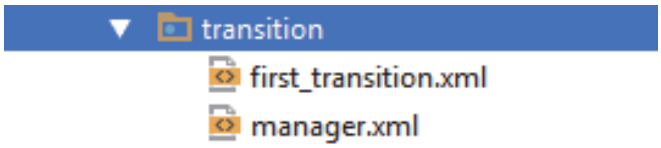


Рис. 13.9. Определение анимации через XML

Затем создать *Transition Manager:*

```
<?xml version="1.0" encoding="utf-8"?>
<transitionManager xmlns:android="http://schemas.android.com/apk/res/android">
    <transition
        android:fromScene="@layout/scene2"
        android:toScene="@layout/scene1"
        android:transition="@transition/first_transition" />
</transitionManager>
```

После этого можно запустить анимацию:

```
int mCurrentScene = 1;
private Scene mScene1,mScene2;
private ViewGroup mSceneRoot;
private TransitionManager transitionManager;

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

    mSceneRoot = (ViewGroup)findViewById(R.id.scene_root);
    mScene1 = Scene.getSceneForLayout(mSceneRoot, R.layout.scene1,this);
    mScene2 = Scene.getSceneForLayout(mSceneRoot, R.layout.scene2,this);

...
transitionManager = TransitionInflater.from(this).inflateTransition-
Manager(R.transition.manager,mSceneRoot);

...
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.action_transition) {
        runTransition();
        return true;
    }
    return super.onOptionsItemSelected(item);
}
private void runTransition() {
    if (mCurrentScene==1){
        transitionManager.go(mScene2);
       mCurrentScene=2;
    }
    else{
        transitionManager.go(mScene1);
        mCurrentScene=1;
    }
    }
```

13.3.5. Physics-based Animations

Fling Animation. Анимация на основе *Fling* использует силу трения, которая пропорциональна скорости объекта. У нее есть начальный импульс, который главным образом определяется скоростью жеста и постепенно замедляется. Анимация заканчивается, когда скорость ее низкая и нет видимых изменений на экране устройства.

Чтобы использовать анимацию на основе физики, нужно добавить библиотеку поддержки в файл *build.gradle*:

```
dependencies { implementation 'com.android.support:support-dynamic-an-
imation:28.0.0'
  }
```

Затем надо создать экземпляр класса *FlingAnimation*, предоставив объект и свойство объекта, которое необходимо анимировать:

```
FlingAnimation fling = new FlingAnimation(view,
                                    DynamicAnimation.SCROLL_X);
```

Далее устанавливается ***скорость анимации***. Начальная скорость показывает, как быстро свойство анимации изменяется вначале (по умолчанию она установлена равной нулю пикселей в секунду). Можно использовать фиксированное значение в качестве начальной скорости или задавать ее на основе скорости жеста касания. Чтобы установить скорость, вызовите метод *setStartVelocity*() и передайте скорость в пикселях в секунду. Метод возвращает объект, на котором установлена скорость.

Также устанавливается ***диапазон значений анимации***. Можно задать минимальное и максимальное значения анимации, когда надо ограничить значение свойства, например альфа (от 0 до 1), применить методы *setMinValue*() и *setMaxValue*().

Метод *setFriction*() позволяет изменить силу трения при анимации. Он определяет, насколько быстро скорость анимации уменьшается. Если не установить трение в начале анимации, то она использует значение трения по умолчанию, равное 1.

Пример ниже иллюстрирует горизонтальное перемещение:

```
FlingAnimation fling = new FlingAnimation(view, DynamicAnimation.SCROLL_X);
fling.setStartVelocity(-velocityX)
        .setMinValue(0)
        .setMaxValue(maxScroll)
        .setFriction(1.1f)
        .start();
```

Spring Animation (пружина). Усилие пружины обладает следующими свойствами: демпфирование и жесткость. В анимации на основе пружины значение и скорость рассчитываются в зависимости от силы пружины, приложенной к каждому кадру. Класс *SpringForce* позволяет настраивать жесткость пружины, ее коэффициент деформирования и конечное положение. Анимация продолжается, пока сила пружины не достигнет равновесия.

Первым шагом является создание экземпляра класса *SpringAnimation* и установка параметров движения:

```
final View img = findViewById(R.id.imageView);
final SpringAnimation springAnim = new SpringAnimation(img,
                                   DynamicAnimation.TRANSLATION_Y, 0);
```

Доступны следующее параметры:

— *ALPHA* – представляет альфа-прозрачность. По умолчанию это значение равно 1 (непрозрачное);

— *TRANSLATION_X*, *TRANSLATION_Y* и *TRANSLATION_Z* – эти свойства управляют тем, где представление находится в виде дельты от его координаты;

— *TRANSLATION_X* – описывает левую координату;

— *TRANSLATION_Y* – описывает верхнюю координату;

— *TRANSLATION_Z* – описывает глубину обзора;

— *ROTATION*, *ROTATION_X* и *ROTATION_Y* – эти свойства управляют вращением в 2D (свойство поворота) и 3D вокруг точки поворота;

— *SCROLL_X* и *SCROLL_Y* – свойства указывают смещение прокрутки левого и верхнего края источника в пикселях;

— *SCALE_X* и *SCALE_Y* – эти свойства управляют 2D-масштабированием вида вокруг его точки поворота;

— *X*, *Y* и *Z* – это основные служебные свойства, которые описывают конечное местоположение представления в его контейнере;

— *X* – сумма левого значения и *TRANSLATION_X*;

— *Y* – сумма верхнего значения и *TRANSLATION_Y*;

— *Z* – сумма значения высоты и *TRANSLATION_Z*.

Есть два способа запустить анимацию: вызывать *start*() или вызывать метод *animateToFinalPosition*(). Оба метода должны быть вызваны в главном потоке.

Метод *animateToFinalPosition*() выполняет две задачи. Устанавливает конечную позицию пружины. Запускает анимацию, если она еще не началась. Поскольку метод обновляет конечную позицию пружины

и запускает анимацию, его можно вызвать в любое время, чтобы изменить ход анимации.

Метод *start*() не устанавливает в начальное значение свойство немедленно. Значение свойства изменяется при каждом импульсе анимации, что происходит перед проходом прорисовки. В итоге все изменения отражаются в следующем кадре, как будто значения устанавливаются немедленно.

```
        final View img = findViewById(R.id.imageView);
        final SpringAnimation anim = new SpringAnimation(img, Dynami-
cAnimation.TRANSLATION_Y);
…
        anim.start();
```

ЛИТЕРАТУРА

1. История версий Android // Википедия [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%82%D0%BE%D1%80%D0%B8%D1%8F_%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B9_Android. – Дата доступа 01.01.2020.

2. Основы создания приложений // Developers [Электронный ресурс]. – Режим доступа: https://developer.android.com/guide. – Дата доступа: 11.12.2019.

3. Дейтел, П. Android для разработчиков / П. Дейтел, Х. Дейтел, А. Уолд. – 3-е изд. – СПб.: Питер, 2016. – 512 с.

4. Android. Программирование для профессионалов / Б. Харди [и др.]. – 2-е изд. – СПб.: Питер, 2016. – 640 с.

5. Гриффитс, Д. Head First. Программирование для Android / Д. Гриффитс. – СПб.: Питер, 2016. – 704 с.

6. Start Android: учебник по Android для начинающих и продвинутых // STARTANDROID [Электронный ресурс]. – Режим доступа: https://startandroid.ru/ru/. – Дата доступа: 01.01.2020.

7. Уроки по разработке андроид-приложений // Fandroid.info [Электронный ресурс]. – Режим доступа: https://www.fandroid.info. – Дата доступа: 11.12.2019.

8. Murphy, Mark L. The Busy Coder’s Guide to Android Development 8.11 / Mark L. Murphy. – CommonsWare, 2018. – 417 с.

9. Darwin, Ian F. Android Cookbook: Problems and Solutions for Android Developers / Ian F. Drawin. – 2nd Edition. – Sebastopol: O'Reilly, 2017. – 838 с.

10. Майер, Р. Android 4. Программирование приложений для планшетных компьютеров и смартфонов / Р. Майер. – М.: Эксмо, 2013. – 814 с.

СОДЕРЖАНИЕ

Учебное издание

Пацей Наталья Владимировна

РАЗРАБОТКА МОБИЛЬНЫХ ПРИЛОЖЕНИЙ

Учебно-методическое пособие

Редактор *Т. Е. Самсанович*
Компьютерная верстка *Е. В. Ильченко*
Корректор *Т. Е. Самсанович*

Издатель:
УО «Белорусский государственный технологический университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/227 от 20.03.2014.
Ул. Свердлова, 13а, 220006, г. Минск.