Министерство образования и науки Российской Федерации
Ярославский государственный университет им. П. Г. Демидова

И. В. Парамонов

Разработка мобильных приложений для платформы Android

Учебное пособие

Рекомендовано
Научно-методическим советом университета
для студентов, обучающихся по направлениям
Прикладная математика и информатика,
Фундаментальная информатика
и информационные технологии

Ярославль
ЯрГУ
2013

УДК 004.4(075.8)
ББК З973.2-018.2я73
П 18

Рекомендовано
Редакционно-издательским советом университета
в качестве учебного издания. План 2013 года.

Рецензенты:
М. А. Васильев, кандидат технических наук, доцент;
Ярославский филиал Физико-технологического
института РАН.

П 18 **Парамонов, И. В.** Разработка мобильных приложений для платформы Android: учебное пособие / И. В. Парамонов; Яросл. гос. ун-т им. П. Г. Демидова. — Ярославль : ЯрГУ, 2013. — 88 с.
ISBN 978-5-8397-0930-0

Учебное пособие посвящено разработке приложений для мобильных устройств, функционирующих под управлением операционной системы Android. Рассмотрены основные API платформы, материал проиллюстрирован большим количеством примеров.

Предназначено для студентов, обучающихся по направлениям 010400.68 Прикладная математика и информатика и 010300.68 Фундаментальная информатика и информационные технологии (дисциплина «Современные мобильные платформы и сервисы», цикл М2), очной формы обучения.

Библиогр.: 12 назв.

УДК 004.4(075.8)
ББК З973.2-018.2я73

ISBN 978-5-8397-0930-0

Оглавление

Введение

В настоящее время операционная система Android является, по-видимому, самой популярной платформой для мобильных устройств. Многообразие и широкое распространение смартфонов и планшетов различных производителей, функционирующих под управлением данной платформы, стимулирует рост рынка мобильных приложений, делая навыки разработки под Android весьма востребованными в современном мире.

Целью настоящего пособия является введение в разработку мобильных приложений для операционной системы Android. При этом стиль изложения материала ориентирован не на охват средств и инструментов, предоставляемых API данной платформы, а на объяснение основных принципов и демонстрацию конкретных возможностей на практике.

Первая глава является введением в платформу. В ней описываются средства разработки, основные компоненты, способы создания, компиляции и развёртывания проекта. Вторая глава посвящена активностям — основным структурным блокам Android-приложений, ответственным за взаимодействие с пользователем. Третья глава целиком посвящена рассмотрению примера приложения, основанного на архитектурном шаблоне MVC. Данная глава заканчивает вводный блок, иллюстрируя на практике сведения из двух первых глав пособия.

Следующие три главы посвящены соответственно компонентам-виджетам платформы, работе с ресурсами и способам хранения данных. Данные темы составляют минимум материала, необходимого любому разработчику приложений для платформы Android. Седьмая глава, как и третья, содержит достаточно обширный пример, связывающий в единое целое материал всех предыдущих глав.

Последние две главы посвящены более сложным вопросам разработки приложений под Android: асинхронному выполнению и провайдерам контента.

1. Основы разработки приложений для ОС Android

1.1. Android SDK. Android SDK — это набор инструментов, позволяющих осуществлять разработку приложений для платформы Android. В его состав входят библиотеки, предоставляющие программисту API платформы Android, утилиты для создания и сборки приложений, управления образами виртуальных устройств и выполнения других действий. Android SDK распространяется бесплатно для всех основных платформ и может быть загружен с сайта Google: *http://developer.android.com/sdk/index.html*.

Android SDK не содержит компилятора языка Java, поэтому для компиляции Android-приложений необходим также набор инструментов Java Development Kit (JDK). Последняя версия JDK от компании Oracle может быть загружена с сайта Oracle по адресу *http://www.oracle.com/technetwork/java/javase/downloads/index.html*.

Для упрощения разработки может быть использована одна из интегрированных сред (Intergrated Development Environment, IDE). В настоящее время поддержка разработки под Android имеется во всех основных средах разработки для Java, в том числе IntelliJ IDEA, NetBeans и Eclipse. Следует отметить, что наличие IDE не является обязательным для разработки, поскольку все необходимые для сборки и развёртывания приложения операции могут быть выполнены средствами командной строки. Настоящее пособие не предполагает использования какой-либо конкретной IDE.

1.2. Менеджер пакетов Android SDK. Android SDK поддерживает разработку под все официальные версии платформы Android с использованием множества библиотек, включая многие библиотеки сторонних разработчиков. Для управления пакетами, обеспечивающими разработку с использованием данных библиотек, используется утилита android. Её можно найти в подкаталоге tools внутри каталога SDK.

Утилита android поддерживает как интерфейс командной строки, так и графический интерфейс пользователя. Для управления библиотеками целесообразно использовать графический интерфейс, доступ

к которому можно получить, запустив утилиту android без параметров. После старта на экране отображается список пакетов, содержащий как установленные в системе пакеты, так и пакеты, доступные для установки. Для установки необходимых пакетов необходимо выбрать их в общем списке и нажать кнопку «Install ...packages». После принятия лицензии начнётся загрузка выбранных пакетов из сети Интернет и их установка.

Пакеты в списке сгруппированы по версиям платформы Android. Каждая версия платформы имеет двойную нумерацию: «коммерческую» и «внутреннюю»: например, Android 4.0.3 соответствует API 15. Первый тип нумерации используется для обозначения поддержки возможностей платформы устройствами на рынке, тогда как в процессе разработки приложений повсеместно используемым является второй тип нумерации.

В каждой из групп в списке содержатся следующие элементы: основной набор API для разработки приложений под данную платформу (SDK Platform), примеры приложений (Samples for SDK), документация на API (Documentation for Android SDK), исходные тексты библиотек платформы (Sources for Android SDK), образы виртуальных устройств для различных архитектур (ARM EABI v7a System Image, Intel x86 Atom System Image и другие). Кроме того, в списке могут быть представлены библиотеки сторонних разработчиков, в том числе предназначенные для определённого класса устройств (например, Google TV Addon).

Для того чтобы начать разработку, необходимо установить основной набор библиотек (SDK Platform), образы виртуальных устройств (System Images) для конкретной платформы, а также общий набор инструментов, не привязанный к версии API (Android SDK Tools, Android SDK Platform-tools в группе Tools). Остальные пакеты являются необязательными.

1.3. Создание проекта. Создать шаблон для нового проекта можно средствами утилиты android, описанной в предыдущем разделе. Для этого используется интерфейс командной строки, например: [1]

```
android create project −−target android−15 −−name HelloWorld \
−−path HelloWorld −−activity MainActivity \
```

[1] Рекомендуется включать пути к подкаталогам tools и platform-tools каталога, содержащего SDK, в переменную окружения **PATH**. Это позволит вызывать утилиту android и другие инструменты командной строки без указания полного пути — так, как это сделано в приведённом примере.

```
--package ru.ac.uniyar.helloworld
```

Назначение ключей приведённой команды:

- **--target** — идентификатор платформы, для которой создаётся проект (список всех доступных платформ определяется набором установленных пакетов Android SDK и может быть получен с помощью команды **android list target**);
- **--name** — имя создаваемого приложения;
- **--path** — путь к каталогу создаваемого проекта;
- **--activity** — имя главной активности создаваемого проекта (соответствует главному экрану приложения; более подробно активности рассмотрены в главе 2);
- **--package** — имя корневого пакета создаваемого приложения (все классы создаваемого приложения будут размещаться внутри данного пакета).

1.4. Структура проекта. В результате выполнения команды из предыдущего пункта в текущем каталоге будет создан каталог проекта с именем HelloWorld, содержащий следующие файлы и каталоги:

- **AndroidManifest.xml** — файл манифеста (более подробно см. п. 1.5);
- **build.xml** — сборочный файл ant;
- **ant.properties**, **project.properties**, **local.properties** — файлы конфигурации ant;
- **src** — каталог, содержащий исходные тексты приложения;
- **res** — каталог, содержащий файлы ресурсов (ресурсы рассмотрены в главе 5).

1.5. Файл манифеста. Файл AndroidManifest.xml является главным конфигурационным файлом Android-приложения. Он содержит определения всех компонентов, из которых состоит приложение, описывает способы взаимодействия между ними, условия сборки проекта, права доступа к различным ресурсам и т. д. Ниже приведён пример простейшего файла манифеста и описаны его основные компоненты.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="ru.ac.uniyar.helloworld"
        android:versionCode="1" android:versionName="1.0">
  <uses-sdk android:minSdkVersion="15" />
```

```
    <uses-feature android:name="android.hardware.location" />
    <uses-permission
        android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <application android:label="@string/app_name"
        android:icon="@drawable/ic_launcher">
      <activity android:name="MainActivity"
          android:label="@string/app_name">
        <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
      </activity>
    </application>
  </manifest>
```

Корневым элементом файла манифеста является элемент **<manifest>** со следующими атрибутами: **package** (имя корневого пакета приложения), **android:versionName** (номер версии приложения, который может видеть пользователь), **android:versionCode** (внутренний номер версии приложения; используется при обновлении приложений, установленных на устройстве).

Элемент **<uses-sdk>** определяет версии SDK, которые могут быть использованы для сборки проекта. В примере выше с помощью атрибута **android:minSdkVersion** указывается, что проект может быть собран с помощью SDK версии API не ниже 15, которая соответствует Android 4.0.3. При попытке сборки проекта с помощью неподходящей версии SDK будет выдано сообщение об ошибке.

Элементы **<uses-feature>** используются для декларации возможностей, которые должны поддерживаться целевым устройством для правильного функционирования приложения. Например, значение **android.hardware.location** означает, что приложению необходимо устройство, обладающее возможностями геолокации. Содержимое элементов типа **<uses-feature>** используется для фильтрации списка приложений в Google Play, доступных для конкретного устройства.

Элементы **<uses-permission>** используются для декларации разрешений на доступ к функциям устройства, которые необходимы приложению для правильного функционирования. Например, разрешение **android.permission.ACCESS_COARSE_LOCATION** запрашивается для доступа к функциям определения местонахождения устройства,

а разрешение **android.permission.INTERNET** — для загрузки содержимого из сети Интернет. При попытке доступа приложения к функциям, требующим разрешений, не перечисленных в файле манифеста, выбрасывается исключение. Когда пользователь загружает приложение из Google Play, он должен просмотреть список разрешений и подтвердить доступ приложения к запрашиваемым функциям.

Элемент **<application>** описывает приложение в целом. Атрибуты **android:label** и **android:icon** этого элемента определяют имя приложения и его иконку, отображаемые в списке приложений на устройстве. Значениями данных атрибутов могут быть как текстовые строки, так и ссылки на ресурсы приложения (как в приведённом выше примере). Работа с ресурсами освещается в главе 5.

Внутри элемента **<application>** размещаются определения компонентов приложения, в данном случае определение главной активности. Более подробно регистрация активностей в файле манифеста рассмотрена в п. 2.3.

1.6. Сборка проекта. Сборка Android-проекта может осуществляться различными способами. В простейшем случае используется утилита ant, входящая в JDK и являющаяся стандартным средством сборки Java-проектов.

Сборочный файл ant обычно называется build.xml и располагается в корневом каталоге проекта. Если создавать проект командой из п. 1.3, то данный файл также будет создан.

В сборочном файле определяются цели, каждая из которых соответствует некоторой операции (например, компиляция, сборка, развёртывание, тестирование проекта). Операции состоят из команд, выполнение которых приводит к достижению указанной цели.

Между целями могут быть установлены зависимости, при этом команды, необходимые для достижения некоторой цели, выполняются только после того, как выполнены команды зависимых целей. При этом поддерживается достаточно гибкий механизм, позволяющий не выполнять некоторые команды, если они уже были выполнены ранее и их повторное выполнение даст тот же результат (например, компиляция модуля не запускается повторно, если этот модуль уже был скомпилирован ранее и его исходный текст не изменялся).

Далее перечислены цели сборочного файла ant, создаваемого утилитой android автоматически:

- **help** — выводит краткую справку по всем автоматически сгенерированным целям;
- **debug** — сборка проекта в редакции, предназначенной для тестирования и отладки приложения разработчиком;
- **release** — сборка проекта в редакции, предназначенной для публикации приложения в Google Play (в этом случае приложение после сборки будет подписано ключом разработчика);
- **install** — установка собранного пакета приложения в запущенном эмуляторе или на устройстве. Данную цель можно использовать только вместе с одной из целей сборки (**debug** или **release**) или указать суффикс, обозначающий версию, которую необходимо установить (**installd** или **installr**);
- **clean** — очистка проекта, удаление всех файлов, содержащих скомпилированный байт-код приложения, а также промежуточные результаты сборки.

Следует иметь в виду, что операции, соответствующие любой из указанных целей, можно изменять для того, чтобы обеспечить выполнение действий, специфичных для конкретного проекта. Также можно добавлять новые цели.

Для сборки проекта и создания пакета следует выполнить команду

```
ant debug
```

Если дополнительно требуется установить собранный пакет в запущенный эмулятор или на подключенное к компьютеру разработчика Android-устройство, то используется команда

```
ant debug install
```

1.7. Вопросы и упражнения для самопроверки:

1. Что такое Android SDK? Какие компоненты он содержит? Какие инструментальные средства можно использовать при разработке приложений на платформе Android?
2. Что такое менеджер пакетов Android? Какие задачи он решает?
3. Какова структура автоматически создаваемого проекта приложения для Android? Какие компоненты создаются и в каких каталогах они размещаются?
4. Что такое файл манифеста? Какова его структура? Какие основные элементы могут встречаться в файле манифеста и для чего они нужны?

5. Что такое ant? Как он используется для сборки приложений? Какие цели содержатся в автоматически сгенерированном файле сборки?

2. Активности и интенты

2.1. Компоненты Android-приложения. Перед тем как перейти к рассмотрению жизненного цикла и возможностей активностей, следует сказать несколько слов об устройстве Android-приложений, поскольку оно существенно отличается от устройства приложений как для настольных компьютеров, так и для других мобильных платформ.

Типичное Android-приложение состоит из компонентов, которые могут относиться к следующим типам:

- активность (activity) — компонент, осуществляющий взаимодействие с пользователем;
- сервис (service) — фоновый процесс;
- провайдер контента (content provider) — компонент, осуществляющий предоставление доступа к данным, находящимся в некотором хранилище;
- слушатель широковещательных сообщений (broadcast receiver) — обработчик некоторого глобального события в операционной системе (например, выключение экрана, низкий заряд батареи и т. д.).

Данные компоненты являются достаточно независимыми друг от друга и имеют чётко определённые интерфейсы для взаимодействия с другими компонентами. Интересная особенность Android-приложений заключается в том, что компоненты различных приложений могут взаимодействовать между собой. Например, в случае, когда приложению необходимо отправить сообщение по электронной почте, оно может вызвать стандартную активность с функционалом почтового клиента, причём после завершения этой активности пользователь вернётся к работе с исходным приложением. И наоборот: программист может разработать собственный почтовый клиент и зарегистрировать его в системе при установке приложения, тем самым разрешив другим приложениям его использование.

Эта особенность несколько размывает само понятие приложения и заставляет рассматривать всю платформу как открытую систему, компоненты которой способны к кооперативному поведению. Взаимодействие компонентов осуществляется посредством отправки

асинхронных сообщений. В приложении каждое из таких сообщений ассоциируется с сущностью, называемой «интент» (intent).

2.2. Интент. В Android-приложениях интент — это объект, инкапсулирующий в себе запрос на выполнение некоторого действия. Интент может включать в себя следующие компоненты:

- действие, которое необходимо выполнить (обязательный компонент);
- набор категорий, позволяющих группировать действия;
- URI, идентифицирующий данные, над которыми необходимо выполнить действие;
- дополнительные параметры (extras), необходимые для выполнения действия.

Следует отметить, что интент, как правило, не содержит в явном виде указания адресата отправляемого сообщения. Вместо этого указывается действие, которое необходимо выполнить. Каждый компонент системы во время установки регистрирует некоторый набор интентов, которые он способен выполнять. Когда соответствующий интент активируется, система находит компонент, способный это действие выполнить, и передаёт ему управление. Если же подходящих компонентов оказывается несколько, выбор предоставляется пользователю.

2.3. Объявление активности в файле манифеста. Каждая активность объявляется в файле манифеста. Фрагмент файла AndroidManifest.xml, содержащий такое определение, приведён ниже.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="ru.ac.uniyar.helloworld"
        android:versionCode="1" android:versionName="1.0">
...
    <activity android:name="MainActivity"
            android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
        <intent-filter>
            <action android:name="android.intent.action.VIEW"></action>
```

```
            <data android:scheme="http"></data>
            <category android:name="android.intent.category.DEFAULT" />
        </intent−filter>
    </activity>
...
</manifest>
```

Главный элемент объявления называется **<activity>**. Параметр **android:name** этого элемента содержит имя активности и используется (совместно с именем корневого пакета приложения) для определения класса, содержащего программный код этой активности. Например, в приведённом выше примере такой класс называется **ru.ac.uniyar.helloworld.MainActivity**, и именно этот класс будет загружен при старте активности. Параметр **android:label** содержит название, которое выводится пользователю, когда активность отображается на экране.

Элементы **<intent-filter>** определяют интенты, которые может обрабатывать данная активность. В приведённом выше примере определены два таких интента. Первый из них обеспечивает возможность отображения приложения в списке всех приложений Android и самостоятельного запуска активности из этого списка. Для этого указывается тип действия **android.intent.action.MAIN** и категория **android.intent.category.LAUNCHER**. Второй элемент **<intent-filter>** указывает, что активность может функционировать как стандартный web-браузер, обрабатывая действие **android.intent.action.VIEW** для типа данных **http**.

Из приведённого примера видно, что для определения действий, которые обрабатывает активность, можно использовать любые компоненты интента. Более подробную информацию о конкретных элементах файла манифеста, позволяющих выполнить это определение, можно найти в документации API платформы.

2.4. Жизненный цикл активности. В архитектуре Android-приложений активности являются короткоживущими компонентами. Они могут автоматически создаваться и уничтожаться в различных ситуациях, например в случае изменения ориентации экрана или нехватки памяти для других приложений. Кроме того, активность может переходить в фон или на передний план, принимать и терять фокус. Для правильной обработки всех этих ситуаций вводится понятие жизненного цикла активности и предоставляются средства,

Рис. 2.1. Жизненный цикл активностей в Android

позволяющие выполнять те или иные действия при переходе между различными фазами этого жизненного цикла.

Жизненный цикл активности включает в себя следующие основные состояния:

- **running/resumed** — активность находится на переднем плане и в фокусе; в этом состоянии пользователь может непосредственно взаимодействовать с приложением посредством графического интерфейса;
- **paused** — активность находится на переднем плане, но не в фокусе, т. е. перекрыта всплывающим окном или окном диалога; пользователь не может непосредственно взаимодействовать с такой активностью;
- **stopped** — активность находится в фоне и не отображается на экране; пользователь не может взаимодействовать с такой активностью.

Переходы между перечисленными состояниями осуществляются по событиям, инициированным пользователем (например, переключением на другую активность), или системным событиям (например, в связи с нехваткой памяти). Каждый переход сопровождается вызовом определённых методов в классе **Activity**, от которого обязаны наследоваться любые пользовательские активности. В свою очередь, в классах-наследниках указанные методы могут переопределяться для того, чтобы должным образом отреагировать на переход между состояниями жизненного цикла. Возможные переходы вместе с соответствующими callback-методами изображены на рис. 2.1. Рассмотрим наиболее важные из упомянутых методов подробнее.

Метод **onCreate()** вызывается непосредственно после создания активности. Здесь осуществляется инициализация интерфейса пользователя, привязка данных к элементам интерфейса, создание потоков и т. д. Парным к данному методу является метод **onDestroy()**, в котором необходимо освободить ресурсы, полученные при вызове **onCreate()**.

Метод **onPause()** вызывается, когда активность теряет фокус в связи с перекрытием её экрана всплывающим окном, диалогом или другой активностью. После возврата из данного метода активность перейдёт

в состояние **paused**. Отметим, что активность, находящаяся в таком состоянии, может быть в любой момент уничтожена операционной системой в случае нехватки оперативной памяти для других, возможно более высокоприоритетных, приложений. В связи с этим в методе **onPause()** необходимо предусмотреть сохранение состояния активности, чтобы иметь возможность восстановить его при перезапуске приложения. Восстановление состояния осуществляется в callback-методе **onResume()**.

В заключение обсуждения жизненного цикла активности обратим внимание ещё на одну особенность платформы Android. Изменение системной конфигурации приложения, включающей, например, поворот экрана или смену локализации, влечёт за собой уничтожение активности и её последующее повторное создание. При этом выполняются все callback-методы, включая **onDestroy()** для уничтожаемой и **onStart()** для вновь создаваемой активности. Для того чтобы отличить рассматриваемую ситуацию от ситуации, когда приложение закрыто пользователем, а затем вновь открыто, предусмотрена пара методов **onSaveInstanceState()** и **onRestoreInstanceState()**. Реализация этих методов в классе **Activity** автоматически сохраняет и восстанавливает состояние всех элементов пользовательского интерфейса. Однако в некоторых сложных случаях может потребоваться переопределение и этих методов для обеспечения корректного функционирования приложения при изменении конфигурации.

Более подробно вопросы корректного сохранения состояния приложения рассмотрены в п. 3.11, а также в главе 6.

2.5. Вызов активности через интент. Как уже отмечалось ранее, вызов активности осуществляется через интент. В программном коде для этого необходимо создать экземпляр класса **Intent**, а затем передать этот экземпляр методу **startActivity()**, определённому в классе **Context** — суперклассе **Activity**. Например:

```
Intent intent = new Intent(Intent.ACTION_VIEW,
    Uri.parse("http://developer.android.com"));
startActivity(intent);
```

Здесь создаётся интент со стандартным действием **ACTION_VIEW** (данное действие ассоциировано с открытием интернет-браузера платформы), а в качестве данных, над которыми необходимо выполнить действие, используется URI веб-страницы *http://developer.android.*

com. После этого платформа находит активность, способную обработать действие **ACTION_VIEW**, и запускает её, передавая указанные данные.

Обычно описанным выше способом осуществляется взаимодействие со стандартными компонентами платформы. При необходимости обращения к частям одного и того же приложения чаще используется упрощённый подход. Идея его состоит в том, что для идентификации требуемой активности используется полное наименование класса данной активности, например:

```
Intent intent = new Intent(this, OtherActivity.class);
intent.setData("http://www.uniyar.ac.ru");
startActivity(intent);
```

В последнем случае при объявлении активности в файле манифеста элемент **<intent-filter>** не указывается:

```
<activity android:name="OtherActivity"
    android:label="@string/other_activity_name" />
```

2.6. Задачи и стек активностей. Под задачей (task) в Android подразумевается набор активностей, вызываемых друг из друга и направленных на удовлетворение одной потребности пользователя. Список всех выполняемых на устройстве задач отображается, когда пользователь нажимает и удерживает кнопку «Home».

Когда пользователь запускает приложение, создаётся новая задача и первая открывшаяся активность запущенного приложения помещается в стек активностей этой задачи. Относительно задачи эта активность называется корневой. Задача существует до тех пор, пока корневая активность не завершится.

Корневая активность может вызвать вторую активность, которая будет помещена в стек текущей задачи поверх корневой. В свою очередь, вторая активность может вызывать третью и т. д. Все эти активности помещаются в стек. При закрытии активности, находящейся на вершине стека (по нажатию кнопки «Back» на Android-устройстве или программно с помощью вызова соответствующего метода), она удаляется из стека, а управление передаётся той активности, которая находилась в стеке непосредственно под удалённой.

При нажатии кнопки «Home» текущая активность переходит в фон, однако весь стек активностей соответствующей задачи сохраняется. Если теперь нажать и удерживать кнопку «Home», то пользова-

тель увидит список активных задач. При выборе любой из них активность, находящаяся на вершине стека, получит фокус, а все остальные активности будут сохраняться в стеке, пока находящиеся выше их активности не будут закрыты.

Следует отметить, что активности, входящие в стек некоторой задачи, могут входить в состав различных приложений. В этом выражается одна из важных концепций Android, декларирующая кооперацию нескольких различных приложений для удовлетворения потребностей пользователя и предоставляющая механизм интентов для осуществления этой кооперации.

2.7. **Получение данных из интента.** Интент, с помощью которого запускается активность, является также посредником между запускающей и запускаемой активностями. Его можно получить, используя вызов метода **getIntent()** в классе дочерней активности. Из получаемого объекта класса **Intent** можно извлечь действие, категории, URI и дополнительные параметры, переданные вызывающей активностью. Для этого используются следующие методы:

```
public String getAction();
public Set<String> getCategories();
public Uri getData();
public Bundle getExtras();
```

Описанная возможность может использоваться как для получения данных, требующих обработки данной активностью, так и для диспетчеризации действий в том случае, когда одна активность способна выполнять несколько различных действий.

2.8. **Возврат результата из активности.** Часто возникает необходимость передать информацию не только от запускающей активности к запускаемой, но и в обратном направлении. Типичный пример — ситуация, когда дочерняя активность используется для ввода данных, необходимых главной активности. В этом случае для запуска дочерней активности необходимо использовать не метод **startActivity()**, а метод **startActivityForResult()**:

```
Intent intent = new Intent(this, EnterAddressActivity.class);
startActivityForResult(intent, 1);
```

Здесь приведён пример запуска гипотетической активности под названием **EnterAddressActivity**. Предположим, что данная активность

запрашивает у пользователя адрес и, при нажатии некоторой кнопки, завершает работу и возвращает введённое значение в главную активность. Для реализации такого возврата значения в обработчике кнопки в классе **EnterAddressActivity** необходим следующий код:

```
String address = ...; // retrieve address from the text field
Intent intent = new Intent();
intent.putExtra("address", address);
setResult(RESULT_OK, intent);
finish();
```

Таким образом, создаётся специальный интент без указания действия и класса, содержащий полученный от пользователя адрес в дополнительном параметре (extra) с ключом «address». Далее с помощью вызова **setResult()** указывается, что вызов активности завершился успешно, после чего дочерняя активность закрывается с помощью вызова метода **finish()**.

Обработка возврата осуществляется в переопределённом методе

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

главной активности. В данный метод в качестве параметров **resultCode** и **data** передаются значения, установленные дочерней активностью, а в качестве **requestCode** — идентификационный параметр из вызова метода **startActivityForResult()** (в приведённом выше примере — 1), используемый для того, чтобы различать возвраты из вызовов различных дочерних активностей.

2.9. Вопросы и упражнения для самопроверки:

1. Из каких компонентов могут состоять Android-приложения? В чём назначение и характерные особенности каждого из компонентов?
2. Что такое интент? Какую роль играют интенты во взаимодействии компонентов на платформе Android?
3. Как объявить активность в файле манифеста? С какой целью необходимо данное объявление?
4. Что такое жизненный цикл активности? Какие особенности платформы заставляют вводить понятие жизненного цикла?
5. Какие callback-методы жизненного цикла активности могут быть переопределены разработчиком? В какие моменты жизненного цикла они будут вызываться? Каково типичное назначение каждого из этих callback-методов?

6. Назовите два способа вызова активности через интент. В чём заключается различие между ними? Когда используется каждый из этих способов?
7. Что такое «задача» в терминах Android? Как задачи связаны с активностями? Как задачи выглядят с точки зрения пользователя?
8. Как получить данные, переданные из одной активности в другую? Как получить результат вызова активности?

3. Пример простого приложения в архитектуре MVC

3.1. Архитектура «модель—вид—контроллер». Приложения для Android, как правило, разрабатываются в соответствии с архитектурным шаблоном «модель—вид—контроллер» (model—view—controller, MVC). Этот шаблон позволяет разделять отдельные слои ответственности приложения таким образом, чтобы упростить поддержку кода и облегчить внесение изменений.

Моделью в терминологии MVC является уровень бизнес-логики, ответственный за хранение и обработку информации. Данный слой не должен делать никаких предположений относительно представления информации или взаимодействия пользователя с приложением. Потенциально это делает классы модели пригодными для повторного применения, в том числе на платформах, отличных от той, на которой они были разработаны.

Вид — это уровень отображения. Он ответствен за отображение данных модели. Классы, принадлежащие данному уровню, как правило, являются виджетами платформы либо унаследованы от них. Вид может обращаться к модели для получения данных, однако модель не должна непосредственно зависеть от вида.

Контроллер — это уровень, ответственный за взаимодействие с пользователем. К данному уровню относятся обработчики событий пользователя и часто код, соединяющий все компоненты системы в единое целое (так называемый glue code). В зависимости от конкретной реализации MVC уровни вида и контроллера могут быть отдельными классами или совмещены. Контроллер, как правило, зависит от модели, однако, как и в случае вида, модель не должна хранить ссылок на конкретные классы контроллера.

3.2. Создание проекта. Начиная с данного раздела, по шагам описывается процесс создания простого приложения для Android, основанного на архитектуре MVC. Данный пример, с одной стороны, даёт краткий обзор стандартных средств, используемых для разработки Android-приложений (таких как ресурсы и обработчики событий жиз-

ненного цикла активности), а с другой стороны, демонстрирует реализацию шаблона проектирования MVC на платформе Android.

Разрабатываемое приложение является очень простым. Оно называется «Счётчик» и состоит из единственного экрана, содержащего две кнопки для увеличения значения счётчика на единицу и сброса его в ноль соответственно. Скриншот приложения приведён на рис. 3.1.

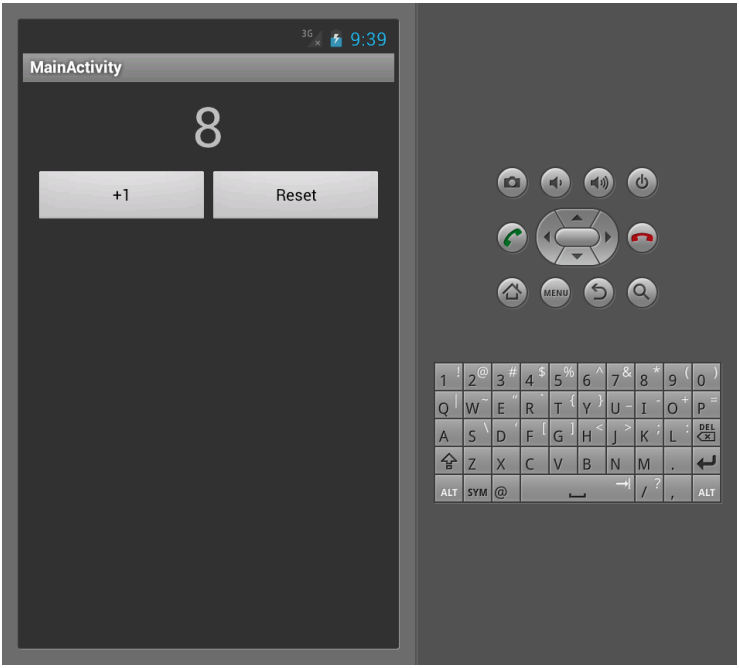

Рис. 3.1. Приложение «Счётчик», запущенное в эмуляторе

Предполагается, что проект создаётся средствами командной строки, как описано в п. 1.3, или с помощью любой интегрированной среды разработки.

Для определённости будем считать, что корневой пакет проекта называется **ru.uniyar.ac.counter** (в нём будут размещены все классы приложения), а класс главной активности имеет имя **MainActivity**. Для справки ниже приведён файл манифеста созданного проекта.

```
<?xml version="1.0" encoding="utf−8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="ru.ac.uniyar.counter"
      android:versionCode="1"
      android:versionName="1.0">
   <application android:label="@string/app_name"
         android:icon="@drawable/ic_launcher">
      <activity android:name="MainActivity"
                     android:label="@string/app_name">
         <intent−filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent−filter>
      </activity>
   </application>
</manifest>
```

3.3. Построение пользовательского интерфейса. На платформе Android пользовательский интерфейс приложения может быть создан двумя способами: декларативно в XML-файле ресурсов и императивно с помощью операторов создания и размещения элементов интерфейса в коде. На практике почти всегда применяется первый способ, поскольку он является более удобным и поддерживается визуальными средствами построения интерфейса.

В нашем примере файл описания интерфейса главной активности создаётся по умолчанию под именем main.xml и располагается в подкаталоге res/layout каталога проекта. Его необходимо отредактировать, добавив метку для отображения значения счётчика и две кнопки для увеличения значения счётчика на единицу и сброса. После редактирования данный файл будет выглядеть следующим образом:

```
<?xml version="1.0" encoding="utf−8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:padding="5dp">
   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
```

```
        android:text="0"
        android:id="@+id/counterText" android:layout_gravity="center"
        android:textSize="48dp"/>
    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" android:padding="5dp">
      <Button
          android:layout_width="0dp"
          android:layout_weight="1"
          android:layout_height="wrap_content"
          android:text="+1"
          android:id="@+id/increaseButton"/>
      <Button
          android:layout_width="0dp"
          android:layout_weight="1"
          android:layout_height="wrap_content"
          android:text="Reset"
          android:id="@+id/resetButton"/>
    </LinearLayout>
</LinearLayout>
```

В приведённом примере корневым элементом XML-документа является элемент **<LinearLayout>**. Он является контейнером и предназначен для размещения вложенных элементов в строку или в столбец. В данном случае, в соответствии со значением свойства **android:orientation**, равным **vertical**, вложенные элементы **<TextView>** и **<LinearLayout>** размещаются в столбец с отступами величиной 5 dp[2] (величина отступа определяется значением атрибута **android:padding**).

Вложенный элемент **<TextView>** представляет собой текстовое поле, отображающее текущее значение счётчика. В соответствии со свойствами элемента данное поле центрировано, текст имеет размер 48 dp и начальное значение «0». Кроме того, текстовое поле имеет идентификатор **counterText**, который необходим для обеспечения возможности обращения к данному полю из кода программы.

Вложенный элемент **<LinearLayout>** располагает свои элементы горизонтально с тем же отступом 5 dp. В роли элементов контейнера

[2] dp (density-independent pixel) — это единица измерения, не зависящая от плотности точек на экране целевого устройства. Такие единицы применяются для того, чтобы соотношение между элементами интерфейса не искажалось при использовании экранов разного размера и разрешения.

выступают кнопки «+1» и «Reset» с идентификаторами **increaseButton** и **resetButton** соответственно. Кнопки делят между собой всё свободное горизонтальное пространство контейнера в пропорции 1:1, так как это определено значением их атрибута **android:layout_weight**. Заметим, что при использовании данного атрибута явно заданная в атрибуте **android:layout_width** ширина кнопок не используется и установлена равной 0.

Приведённый пример даёт представление о том, каким образом описывается размещение элементов в XML-файле ресурсов. Более подробную информацию по данному вопросу можно найти в документации. Несмотря на то, что такие файлы ресурсов обычно генерируются визуальным построителем пользовательского интерфейса, для успешного поиска ошибок и эффективного построения сложных интерфейсов полезно понимать их устройство и на уровне XML-разметки.

3.4. Загрузка пользовательского интерфейса из XML-файла и доступ к его компонентам. После того, как XML-файл, описывающий пользовательский интерфейс активности, сформирован, его необходимо загрузить из программного кода. Обычно это осуществляется в методе **onCreate()** класса активности:

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

Первая строка данного метода вызывает метод **onCreate()** суперкласса — это обязательное действие для всех callback-методов жизненного цикла. Вторая строка указывает, что описание пользовательского интерфейса активности необходимо загрузить из файла ресурсов layout/main.xml. Местоположение и имя файла ресурсов специфицируется с помощью специального идентификатора (в данном случае **R.layout.main**), который получается в виде константы из автоматически генерируемого класса **R**. Данный класс пересоздаётся при каждой сборке проекта, а при использовании интегрированной среды ещё и после внесения изменений в файлы ресурсов. Файл, содержащий класс **R**, располагается в подкаталоге gen каталога проекта и не должен храниться в системе контроля версий.

Кроме того, в методе **onCreate()** класса активности, как правило, запрашиваются ссылки на Java-объекты элементов интерфейса, к кото-

рым в дальнейшем потребуется доступ из программного кода. Данные ссылки получаются с помощью вызова метода **findViewById()** и обычно сохраняются в полях класса активности.

С учётом всего сказанного файл MainActivity.java может выглядеть следующим образом:

```
public class MainActivity extends Activity {
    private TextView counterText;
    private Button increaseButton, resetButton;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        counterText = (TextView) findViewById(R.id.counterText);
        increaseButton = (Button) findViewById(R.id.increaseButton);
        resetButton = (Button) findViewById(R.id.resetButton);
    }
}
```

Заметим, что для определения элементов интерфейса, которые требуется получить, используются идентификаторы, заданные в качестве значений свойства **android:id** соответствующих элементов XML-файла.

3.5. Обработка событий элементов интерфейса пользователя. В рассматриваемом примере необходимо обработать события от двух кнопок пользовательского интерфейса «+1» и «Reset» и предпринять необходимые действия в ответ на касание этих кнопок. Существуют два способа решить данную задачу.

Первый способ состоит в добавлении обработчика кнопки непосредственно из программного кода. Для этого необходимо выполнить следующие действия:

1) определить класс, реализующий интерфейс слушателя **Button.OnClickListener**;
2) реализовать метод **onClick()**, который определяется данным интерфейсом;
3) создать объект класса-слушателя и установить его в качестве обработчика касания кнопки.

Обычно в качестве обработчика выступает вложенный анонимный класс. Его использование позволяет реализовать все три действия в ко-

де наиболее компактно. Рассмотрим использование анонимного класса для обработки события — касания кнопки «+1»:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  // ...
  increaseButton.setOnClickListener(new Button.OnClickListener() {
    @Override
    public void onClick(View v) {
      onIncreaseButtonClick(v);
    }
  });
}

private void onIncreaseButtonClick(View v) {
  // handler code here
}
```

Поскольку синтаксис вложенных классов является достаточно громоздким, собственно код обработчика события вынесен в отдельный метод **onIncreaseButtonClick()** класса **MainActivity**. Рекомендуется придерживаться данного подхода, поскольку он делает код более удобочитаемым.

Второй способ обработки события является декларативным. Он состоит в том, что имя метода-обработчика события в классе **MainActivity** просто указывается в соответствующем атрибуте элемента **<Button>** в файле main.xml:

```
<Button
  android:layout_width="0dp" android:layout_weight="1"
  android:layout_height="wrap_content" android:text="+1"
  android:id="@+id/increaseButton"
  android:onClick="onIncreaseButtonClick" />
```

Необходимо отметить, что в этом случае метод-обработчик в классе **MainActivity** должен быть объявлен открытым (public) и принимать те же аргументы, что и соответствующий метод-обработчик в классе **Button.Listener**:

```
public void onIncreaseButtonClick(View v) {
  // handler code here
}
```

Данный метод обработки является более простым, он не требует внесения изменений в код метода **onCreate()**, однако у него есть небольшой недостаток: в случае ошибки в определении метода-обработчика (опечатка в имени, неправильные типы аргументов и т. д.) эта ошибка будет обнаружена лишь при попытке вызова обработчика во время выполнения программы. Первый способ лишён такого недостатка, и при его использовании все ошибки подобного рода будут обнаружены уже на стадии компиляции.

3.6. Модель счётчика. Для решения нашей задачи, в соответствии с архитектурой MVC, необходимо определить класс уровня модели, который будет хранить данные и содержать бизнес-логику приложения, сводящуюся в данном случае к выполнению операций приращения и сброса счётчика. В простейшем случае такая модель может выглядеть следующим образом:

```
package ru.ac.uniyar.counter;
public class Counter {
  private int value = 0;
  public int getValue() { return value; }
  public void increase() { value++; }
  public void reset() { value = 0; }
}
```

3.7. Встраивание модели в контроллер. Поскольку представленная в предыдущем пункте модель является пассивной (т. е. не может сообщить виду и контроллеру о своих изменениях), в такой разновидности MVC контроллер оказывается ответственным за обновление вида при изменении модели. В роли контроллера в рассматриваемом примере выступает активность **MainActivity**, а в роли вида — текстовое поле **counterText** этой активности.

Для начала добавим поле **counter** в класс **MainActivity** и проинициализируем его:

```
public class MainActivity extends Activity {
  // ...
  private Counter counter = new Counter();
  // ...
}
```

Далее укажем имена двух методов-обработчиков касаний кнопок в файле main.xml:

```
<Button android:id="@+id/increaseButton"
  android:layout_width="0dp" android:layout_weight="1"
  android:layout_height="wrap_content" android:text="+1"
  android:onClick="onIncreaseButtonClick" />
<Button android:id="@+id/resetButton"
  android:layout_width="0dp" android:layout_weight="1"
  android:layout_height="wrap_content" android:text="Reset"
  android:onClick="onResetButtonClick" />
```

Наконец, определим реализации этих методов в классе **MainActivity**:

```
public void onIncreaseButtonClick(View v) {
  counter.increase();
  counterText.setText(String.valueOf(counter.getValue()));
}
public void onResetButtonClick(View v) {
  counter.reset();
  counterText.setText(String.valueOf(counter.getValue()));
}
```

Приведённые обработчики обращаются к модели для того, чтобы изменить данные требуемым образом (увеличить значение счётчика или сбросить его), а также для получения текущего значения. Последнее действие необходимо для обновления содержимое вида.

Приложение-счётчик готово. После компиляции, сборки и развёртывания на экране эмулятора или Android-устройства должен отображаться экран, подобный изображённому на рис. 3.1.

3.8. Активная модель. Реализация приложения-счётчика, описанная в предыдущих разделах, имеет недостаток, связанный с необходимостью отслеживать все изменения модели в контроллере и обновлять вид в соответствии с этими изменениями. В сложных приложениях это может стать серьёзной проблемой, выражающейся в существенной перегрузке кода контроллера. Для её решения можно использовать вместо пассивной модели активную, т. е. такую, которая сможет сама уведомлять слушателей о своих изменениях.

Реализация уведомлений в активной модели похожа на реализацию обработчиков событий от компонентов пользовательского интерфейса Android. Рассмотрим код активной модели:

```
package ru.ac.uniyar.counter;
```

```
public class Counter {
    public interface OnModificationListener {
        void onModification(Counter sender);
    }
    private int value = 0;
    private OnModificationListener listener = null;
    public void setOnModificationListener(OnModificationListener listener) {
        this.listener = listener;
    }
    public int getValue() {
        return value;
    }
    public void increase() {
        value++;
        if (listener != null) { listener.onModification(this); }
    }
    public void reset() {
        value = 0;
        if (listener != null) { listener.onModification(this); }
    }
}
```

Внутри класса модели определяется специальный интерфейс слушателя **Counter.OnModificationListener**. Этот интерфейс должен быть реализован объектом-слушателем, которому требуется получать уведомления (в качестве такого объекта может выступать класс, относящийся к виду или контроллеру в архитектуре MVC). Этот объект должен также вызвать метод **setOnModificationListener()** модели для того, чтобы зарегистрировать себя в качестве слушателя. В момент регистрации модель сохраняет переданную ссылку на слушателя в поле **onModificationListener**.

Каждый раз, когда происходит изменение значения поля **value**, модель вызывает интерфейсный метод **onModification()** на сохранённом объекте-слушателе, тем самым посылая уведомление, на которое слушатель может отреагировать, например выполнив перерисовку изображения на экране.

3.9. Модификация класса активности для использования активной модели. При использовании активной модели необходимо внести три изменения в код класса активности: зарегистрировать обработ-

чик изменений модели, реализовать код этого обработчика и удалить принудительное обновление вида при изменении модели.

Реализуем обработчик модели в виде вложенного класса, передающего управление соответствующему методу класса активности:

```
@Override
public void onCreate(Bundle savedInstanceState)
{
  // ...
  counter.setOnModificationListener(new Counter.OnModificationListener() {
    @Override
    public void onModification(Counter sender) {
      updateCounterView();
    }
  });
}
```

Легко видеть, что применённый подход полностью совпадает с первым методом обработки событий из п. 3.5, осуществляя передачу обработки событий методу **updateCounterView()** класса **MainActivity**. Реализация этого метода выглядит следующим образом:

```
public void updateCounterView() {
  counterText.setText(String.valueOf(counter.getValue()));
}
```

Осталось удалить операторы обновления вида из обработчиков кнопок «+1» и «Reset», поскольку обновление теперь будет выполняться методом **updateCounterView()**, вызываемым автоматически при изменении модели.

После удаления ненужного кода указанные выше обработчики будут выглядеть следующим образом:

```
public void onIncreaseButtonClick(View v) {
  counter.increase();
}
public void onResetButtonClick(View v) {
  counter.reset();
}
```

На этом замена пассивной модели на активную завершена. Можно убедиться, что построенное в данном пункте приложение работает в точности так же, как и приложение с пассивной моделью.

3.10. Преимущества и недостатки активной и пассивной модели. Проведём сравнение двух различных реализаций одной и той же модели и попытаемся на его основе сформулировать преимущества и недостатки использования этих моделей, а также определить, в каких случаях следует отдавать предпочтение одной из них.

Рассматривая классы пассивной (п. 3.6) и активной (п. 3.8) моделей, легко убедиться, что активная модель является существенно более сложной по сравнению с пассивной, дополнительно включая в себя поле для хранения ссылки на слушателя, метод для установки этого поля и код уведомления слушателя во всех случаях изменения модели. Кроме того, при использовании активной модели необходим код регистрации слушателя в методе **onCreate()** класса активности, который отсутствует в случае пассивной модели.

Однако, несмотря на более высокую сложность реализации, есть у активной модели и существенное преимущество: благодаря её использованию нет необходимости заботиться об обновлении вида при изменении модели. Данное свойство становится особенно ценным, когда изменения модели не локализованы в коде. В качестве примера рассмотрим приложение, позволяющее пользователю играть в некоторую игру по сети. Изменения модели (игрового поля) в этом случае могут происходить как через графический интерфейс пользователя, так и через модуль сетевого взаимодействия. Если в данной ситуации не использовать активную модель, код сетевого взаимодействия может оказаться зависимым от кода, осуществляющего отображение, нарушая тем самым требования архитектурного шаблона MVC. При использовании же активной модели код изменения модели оказывается отделён от кода обновления вида, а установка слушателей производится в контроллере, что делает приложение концептуально более понятным за счёт чёткого разделения уровней модели, вида и контроллера.

Из сказанного выше можно сформулировать следующие соображения относительно выбора типа модели. Если изменения модели локализованы в коде и не составляют большого объёма, то пассивной модели может быть достаточно. В противном случае следует применять активную модель, и дополнительные усилия по написанию кода работы со слушателями будут скомпенсированы повышением прозрачности кода и его более высокой надёжностью.

3.11. Обработка смены ориентации экрана. Разработанное приложение имеет дефект, который можно легко обнаружить, если повернуть устройство, на котором запущено приложение, или имитировать поворот в эмуляторе с помощью комбинаций клавиш Ctrl+F11 или Ctrl+F12. Нетрудно заметить, что при этом значение счётчика сбрасывается в ноль. Это происходит потому, что при повороте активность уничтожается и создаётся заново, причём новый экземпляр класса-активности создаёт новую модель с нулевым значением счётчика.

Чтобы устранить данный дефект, необходимо сохранять состояние модели перед уничтожением активности и восстанавливать его после создания нового экземпляра. Для сохранения состояния в классе **Activity** предусмотрен метод

```
protected void onSaveInstanceState(Bundle outState);
```

Данный метод вызывается, когда экземпляр активности уничтожается, но будет пересоздан, и не вызывается при нормальном завершении работы приложения (например, если пользователь нажимает кнопку «Back»). Передаваемый методу **onSaveInstanceState()** в качестве аргумента объект класса **Bundle** предназначен для сохранения данных, которые потребуются активности для восстановления состояния после пересоздания. Само восстановление может быть осуществлено либо в методе

```
protected void onRestoreInstanceState(Bundle savedInstanceState);
```

либо в методе **onCreate()**. Обоим методам передаётся объект класса **Bundle** с данными, сохранёнными методом **onSaveInstanceState()**. В случае, когда восстановление данных не требуется (т. е. произошло первоначальное создание класса активности, а не его пересоздание в связи с изменением конфигурации), метод **onRestoreInstanceState()** не вызывается, а методу **onCreate()** в качестве значения параметра **savedInstanceState** передаётся null.

Поскольку реализация класса **Counter** из п. 3.8 предусматривает отсчёт значений только от нуля, необходимо создать конструктор, позволяющий инициализировать модель произвольным значением. Конструктор по умолчанию при этом необходимо определить явно. В итоге в класс **Counter** необходимо внести следующие изменения:

```
public class Counter {
  private int value;
```

```
    public Counter() { value = 0; }
    public Counter(int initialValue) { value = initialValue; }
    // ...
}
```

Теперь добавим в класс **MainActivity** метод **onSaveInstanceState()**, сохраняющий состояние модели:

```
@Override
protected void onSaveInstanceState(Bundle bundle) {
    super.onSaveInstanceState(bundle);
    bundle.putInt("counterValue", counter.getValue());
}
```

Осталось внести изменения в метод инициализации активности **onCreate()**. Для полноты изложения приведём код данного метода целиком:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    if(savedInstanceState != null) {
        counter = new Counter(savedInstanceState.getInt("counterValue"));
    }
    counterText = (TextView) findViewById(R.id.counterText);
    updateCounterView();
    counter.setOnModificationListener(new Counter.OnModificationListener() {
        @Override
        public void onModification(Counter sender) {
            updateCounterView()
        }
    });
}
```

Основных изменений в коде два. Первое из них состоит в создании объекта модели с помощью конструктора с параметром в ситуации пересоздания активности (это определяется путём сравнения параметра **savedInstanceState** с null). В противном случае в поле **counter** остаётся объект, созданный конструктором по умолчанию. Второе изменение заключается в вызове метода **updateCounterView()**. Это необходимо, поскольку значение счётчика при старте может быть ненулевым, что

не согласуется со значением по умолчанию для текстового поля, которое это значение отображает. Приведённый вызов осуществляет принудительную синхронизацию текстового поля с состоянием модели, решая эту проблему.

3.12. Вопросы и упражнения для самопроверки:

1. Что такое архитектурный шаблон MVC? Из каких компонентов состоят системы, основанные на данном шаблоне?
2. Опишите допустимые способы связи и взаимодействия компонентов в рамках MVC. Обоснуйте, почему именно эти способы взаимодействия разрешены, а другие — нет.
3. Охарактеризуйте способ построения пользовательского интерфейса, применяемый в Android-приложениях.
4. Опишите, каким образом можно загрузить описание пользовательского интерфейса из кода и как можно получить доступ к отдельным виджетам.
5. Опишите два способа обработки событий в Android. Укажите достоинства и недостатки каждого из них.
6. Определите активную и пассивную модели в терминах архитектурного шаблона MVC. Осветите достоинства и недостатки каждого из типов моделей.
7. Опишите, как правильно обрабатывать событие поворота экрана пользователем. Что происходит при повороте с точки зрения жизненного цикла активности?
8. Создайте проект приложения, описанный в этой главе. Скомпилируйте приложение и запустите его в эмуляторе или на реальном устройстве.

4. Класс View и его возможности

4.1. Назначение класса View. Класс **View** является суперклассом всех классов-виджетов в Android, включая **TextView**, **ImageView**, **Button** и т. д. Каждый экземпляр класса **View** ответствен за отрисовку некоторой прямоугольной области на экране, а также за обработку событий, связанных с этой областью. При разработке приложений под Android, как правило, используются готовые библиотечные субклассы класса **View**. Однако в некоторых случаях бывают необходимы компоненты, имеющие специфический внешний вид и поведение. Подобные компоненты можно легко реализовать, унаследовав собственный класс от класса **View** и переопределив методы, ответственные за отрисовку и/или обработку событий.

Из всех событий, обрабатываемых классом **View**, наиболее важными представляются события, возникающие при взаимодействии пользователя с областью, за которую ответствен **View**, события передачи фокуса, нажатия на клавиши и отрисовки. Рассмотрим их более подробно.

4.2. События касания экрана. Когда пользователь касается области на экране, занимаемой конкретным экземпляром класса **View**, происходит событие, которое обрабатывается методом

```
public boolean onTouchEvent(MotionEvent event);
```

Кроме переопределения данного метода, существует ещё возможность обработки события касания с помощью класса-слушателя типа **View.OnTouchListener**. Метод-обработчик данного события следующий:

```
public boolean onTouch(View v, MotionEvent event);
```

Оба метода принимают в качестве параметра объект класса **MotionEvent**, который описывает детали произведённого касания. Основные свойства[3] этого класса:

[3] Напомним, что для получения значений свойств необходимо вызывать соответствующие методы доступа, например **getX()**, **getY()** и т. д.

- **x**, **y** — координаты касания в собственной системе координат виджета;
- **action** — тип события (см. объяснение ниже);
- **pressure** — сила давления на экран (вещественное число от 0.0 до 1.0; для экранов, которые не поддерживают определение силы давления, всегда равно 1.0);
- **size** — размер области касания.

Когда пользователь касается экрана, генерируется событие типа **MotionEvent.ACTION_UP**, а когда отнимает палец от экрана — событие типа **MotionEvent.ACTION_DOWN**. Если пользователь ведёт пальцем по экрану, то дополнительно генерируется серия событий типа **MotionEvent.ACTION_MOVE**, каждое из которых содержит описание фрагмента траектории, по которой двигался палец пользователя. Для получения точек, входящих в траекторию, используются методы

```
public final int getHistorySize();
public final float getHistoricalX(int index);
public final float getHistoricalY(int index);
public final float getHistoricalPressure(int index);
public final float getHistoricalSize(int pos);
```

класса **MotionEvent**. Заметим, что группировка точек во фрагменты траекторий сделана для повышения производительности, чтобы не генерировать отдельных событий для каждой точки.

В качестве примера обработки событий касания экрана приведём код метода **OnTouchEvent()**, выводящий в журнал (log) описание всех возникающих событий:

```
@Override
public boolean onTouch(View v, MotionEvent event) {
  float x = event.getX(), y = event.getY();
  switch (event.getAction()) {
    case MotionEvent.ACTION_DOWN:
      Log.d(getClass().getSimpleName(), "down: " + x + ", " + y);
      break;
    case MotionEvent.ACTION_UP:
      Log.d(getClass().getSimpleName(), "up: " + x + ", " + y);
      break;
    case MotionEvent.ACTION_MOVE:
      Log.d(getClass().getSimpleName(), "move: ");
      for (int i = 0; i < event.getHistorySize(); i++) {
```

```
            Log.d(getClass().getSimpleName(), "−−−−− " +
                event.getHistoricalX(i) + ", " + event.getHistoricalY(i));
        }
        Log.d(getClass().getSimpleName(), "−−−−− " + x + ", " + y);
        break;
    }
    return true;
}
```

Пример вывода приложения:

```
18:32:30.270: DEBUG/MainActivity(1478): down: 241.0, 427.0
18:32:30.412: DEBUG/MainActivity(1478): up:   241.0, 427.0
18:32:31.740: DEBUG/MainActivity(1478): down: 137.0, 165.0
18:32:31.760: DEBUG/MainActivity(1478): move:
18:32:31.770: DEBUG/MainActivity(1478): ----- 137.0, 154.0
18:32:31.770: DEBUG/MainActivity(1478): move:
18:32:31.770: DEBUG/MainActivity(1478): ----- 139.0, 135.0
18:32:31.770: DEBUG/MainActivity(1478): ----- 141.0, 133.0
18:32:31.800: DEBUG/MainActivity(1478): move:
18:32:31.810: DEBUG/MainActivity(1478): ----- 147.0, 141.0
18:32:31.990: DEBUG/MainActivity(1478): up:   147.0, 141.0
```

4.3. События клавиатуры. Обработка событий от клавиатуры встречается достаточно редко, поскольку многие Android-устройства не имеют аппаратной клавиатуры. Однако данная возможность может потребоваться для обработки стандартных кнопок, таких как «Menu» и «Back».

Для обработки событий от клавиатуры необходимо переопределить методы

```
public boolean onKeyDown(int keyCode, KeyEvent event);
public boolean onKeyUp(int keyCode, KeyEvent event);
```

или зарегистрировать слушателя типа **View.OnKeyEventListener** и определить метод

```
boolean onKey(View v, int keyCode, KeyEvent event);
```

Как и в случае с событиями касания экрана, свойство **action** объекта **KeyEvent** позволяет определить тип события:

- **KeyEvent.ACTION_DOWN** — нажатие клавиши;
- **KeyEvent.ACTION_UP** — её отпускание;
- **KeyEvent.ACTION_MULTIPLE** — автоповтор.

Параметр **keyCode** содержит код нажатой клавиши. Например, значение **KeyEvent.KEYCODE_DPAD_LEFT** соответствует кнопке «влево» джойстика телефона, а значение **KeyEvent.KEYCODE_MENU** — кнопке «Menu». Остальные значения кодов можно найти в документации. Отметим, что события от клавиатуры передаются только тому виджету, который в настоящий момент имеет фокус ввода. В число таких виджетов входит, например, компонент **EditText** и не входит компонент **TextView**. Для того чтобы пользовательский компонент мог принимать фокус ввода, необходимо установить его свойство **focusable** равным true.

4.4. Правила обработки событий вдоль иерархии виджетов. Любой виджет может содержать другие виджеты внутри той области, за которую он ответствен. Иерархия включения может быть сформирована непосредственно в программном коде или путём вложения элементов в XML-файле, описывающем пользовательский интерфейс (см. п. 3.3). Как правило, на практике используется второй способ.

Иерархия виджетов имеет существенное значение для обработки событий. Действуют специальные правила, определяющие обработку событий вдоль иерархии:

- в первую очередь событие передаётся наиболее вложенному (листовому в дереве вложения) виджету, ответственному за область, в которой произошло событие;
- если виджет не определяет собственного обработчика для произошедшего события, оно передаётся для обработки родительскому в терминах иерархии включения виджету; иначе вызывается обработчик дочернего виджета;
- если обработчик события возвращает true, то считается, что виджет обработал событие, и дальнейшая обработка не требуется;
- если обработчик возвращает false, то виджет обработал событие, но требуется также продолжить обработку этого события родительским виджетом.

В заключение отметим, что данные правила являются достаточно гибкими, позволяя как переопределять способ обработки событий во вложенных виджетах, так и связывать в цепочку обработчики различных виджетов иерархии.

4.5. Рисование на виджетах. В большинстве случаев причиной создания собственного виджета является необходимость определения

нестандартного способа отрисовки содержимого. Примерами могут служить виджеты, содержащие поля для всевозможных игровых приложений, построения диаграмм, обработки изображений и т. д.

Для того чтобы определить собственный способ отрисовки содержимого виджета, необходимо переопределить метод

```
protected void onDraw(Canvas canvas);
```

Параметр **canvas**, передаваемый данному методу, представляет собой «канву» — объект, инкапсулирующий графический контекст и предоставляющий методы для рисования графических примитивов и оперирования системой координат. При этом параметры рисования задаются объектами класса **Paint**. Графические возможности платформы Android очень обширны, а детали их использования можно найти в документации, поэтому ограничимся рассмотрением примера, демонстрирующего применение некоторых из упомянутых возможностей.

Данный пример представляет собой виджет, отображающий циферблат часов. Будем рассматривать методы соответствующего класса последовательно, давая при этом необходимые комментарии.

```
public class ClockView extends View {
  public ClockView(Context context, AttributeSet attrs) {
    super(context, attrs);
  }
```

Класс виджета-циферблата наследуется от класса **View** и определяет конструктор, вызывающий конструктор суперкласса. Это необходимо для правильного создания экземпляра класса по его описанию из XML-файла.

```
  @Override
  protected void onDraw(Canvas canvas) {
    int size = Math.min(getWidth(), getHeight()) / 2;
    canvas.translate(getWidth() / 2, getHeight() / 2);
    Paint paint = new Paint();
    paint.setAntiAlias(true);
    drawClockFace(canvas, size, paint);
    drawScale(canvas, size, paint);
    drawNumbers(canvas, size, paint);
```

```
        drawHands(canvas, size, paint);
    }
```

Переопределённый метод **onDraw()** определяет размеры будущего циферблата и перемещает начало координат его в центр. После этого создаётся объект класса **Paint** и устанавливается параметр, обеспечивающий сглаживание графики, выводимой на экран. Дальнейшие шаги отрисовки выполняются отдельными методами.

```
    private void drawClockFace(Canvas canvas, int size, Paint paint) {
        paint.setStyle(Paint.Style.FILL);
        paint.setColor(Color.DKGRAY);
        paint.setStrokeWidth(4);
        canvas.drawCircle(0, 0, size, paint);
    }
```

Метод **drawClockFace()** рисует пустой циферблат тёмно-серого цвета с помощью метода **drawCircle()** канвы.

```
    private void drawScale(Canvas canvas, int size, Paint paint) {
        paint.setStyle(Paint.Style.STROKE);
        paint.setColor(Color.YELLOW);
        paint.setStrokeWidth(3);
        for (int i = 0; i < 12; i++) {
            canvas.drawLine(size - 20, 0, size, 0, paint);
            canvas.rotate(30);
        }
    }
```

Метод **drawScale()** рисует 12 делений шкалы циферблата линиями жёлтого цвета толщины 3. Для упрощения кода после отрисовки каждого деления система координат поворачивается на 30 градусов. Это позволяет фактически рисовать все деления, используя одни и те же координаты, но в разных системах координат. По окончании отрисовки система координат вернётся в исходное положение автоматически.

```
    private void drawNumbers(Canvas canvas, int size, Paint paint) {
        paint.setTextSize(28);
        paint.setTextAlign(Paint.Align.CENTER);
        canvas.drawText("12", 0, -size + 60, paint);
```

```
        canvas.drawText("6", 0, size − 60 + textSize("6", paint).y, paint);
        paint.setTextAlign(Paint.Align.RIGHT);
        canvas.drawText("3", size − 40, textSize("3", paint).y / 2, paint);
        paint.setTextAlign(Paint.Align.LEFT);
        canvas.drawText("9", −size + 40, textSize("9", paint).y / 2, paint);
    }
```

Метод **drawNumbers()** рисует числовые метки на шкале. Для простоты ограничимся метками, кратными трём. Метод **setTextAlign()** класса **Paint** используется для того, чтобы обеспечить горизонтальное выравнивание выводимого текста относительно точки, задаваемой параметром **drawText()**. К сожалению, в Android API нет аналогичных средств, обеспечивающих вертикальное выравнивание, и приходится решать эту задачу путём непосредственного вычисления координат. Для этого используется вспомогательный метод **textSize()**:

```
    private static Point textSize(String text, Paint paint) {
        Rect bounds = new Rect();
        paint.getTextBounds(text, 0, text.length(), bounds);
        return new Point(bounds.width(), bounds.height());
    }
```

Заметим, что вспомогательный метод потребовался, потому что метод **getTextBounds()**, с помощью которого определяется фактический размер текста, не возвращает значения, а имеет выходной параметр.

```
    private void drawHands(Canvas canvas, int size, Paint paint) {
        canvas.rotate(−90);
        canvas.save();
        canvas.rotate(1.9f * 360 / 12);
        drawArrow(canvas, size / 3, paint);
        canvas.restore();
        canvas.rotate(1.9f * 360);
        drawArrow(canvas, size * 5 / 8, paint);
    }
```

Метод **drawHands()** рисует стрелки часов. Для рисования стрелок в нужных позициях используется тот же подход, что и при рисовании делений: система координат поворачивается так, чтобы стрелка в новой системе координат была горизонтальной. Для простоты примера отображается всегда фиксированное время — 1:54 (1.9 часа).

```
    private void drawArrow(Canvas canvas, int length, Paint paint) {
        Path path = new Path();
        path.moveTo(0, 0);
        path.rLineTo(length, 0);
        path.rLineTo(0, 10);
        path.rLineTo(20, −10);
        path.rLineTo(−20, −10);
        path.rLineTo(0, 10);
        paint.setStyle(Paint.Style.FILL_AND_STROKE);
        paint.setStrokeWidth(2);
        paint.setColor(Color.RED);
        canvas.drawPath(path, paint);
    }
} // class ClockView
```

Последний метод **drawArrow()** рисует стрелку часов, демонстрируя следующую технику: сначала создаётся объект-путь (класс **Path**), состоящий из последовательности отрезков, а затем этот пусть отрисовывается и заливается цветом (т. к. используется стиль **Paint.Style.FILL_AND_STROKE**) с помощью вызова метода **drawPath()**.

В заключение раздела приведём содержимое файла res/layout/main.xml, описывающего пользовательский интерфейс главной активности с виджетом типа **ClockView**:

```
<?xml version="1.0" encoding="utf−8"?>
<view android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    class="ru.ac.uniyar.clockviewdemo.ClockView"
    xmlns:android="http://schemas.android.com/apk/res/android" />
```

4.6. Вопросы и упражнения для самопроверки:

1. Каково назначение класса **View**? В каких случаях требуется создавать субклассы этого класса?
2. Опишите, что необходимо сделать, чтобы обработать события касания экрана устройства пользователем.
3. Опишите, что необходимо сделать, чтобы обработать события от клавиатуры Android-устройства.
4. Сформулируйте правило обработки событий вдоль иерархии виджетов. С какой целью это правило установлено?

5. Назовите основные классы, позволяющие выполнять рисование на произвольных виджетах. Осветите возможности этих классов. Проиллюстрируйте эти возможности примерами.
6. Создайте проект приложения, описанный в п. 4.5. Скомпилируйте приложение и запустите его в эмуляторе или на реальном устройстве.

5. Работа с ресурсами

5.1. Понятие ресурсов и их назначение. Ресурсы в Android — это статические данные (например, текст, изображения, описание пользовательского интерфейса), являющиеся частью приложения. Ресурсы размещаются разработчиком внутри проекта в виде файлов и переносятся в apk-пакет приложения автоматически во время сборки.

Использование ресурсов преследует две основные цели.

1. Отделение данных от кода. При использовании ресурсов данные хранятся отдельно от кода в декларативном виде, поэтому их легко изменять, причём изменения могут вносить не программисты, а, например, дизайнеры пользовательского интерфейса или переводчики. Изменение ресурсов не требует перекомпиляции приложения — необходима лишь его пересборка.
2. Обеспечение вариативности используемых ресурсов в зависимости от конфигурации. Ресурсы позволяют реализовать поддержку различных типов ориентации экрана и различных языков интерфейса пользователя без дополнительных затрат со стороны программиста, так как ресурсы, подходящие для текущей конфигурации системы, загружаются автоматически.

5.2. Классификация ресурсов. В Android выделяются следующие основные типы ресурсов.

- **Layout** — файлы в формате XML, описывающие расположение элементов интерфейса пользователя. Пример такого файла уже рассматривался ранее в п. 3.3.
- **Menu** — файлы в формате XML, описывающие компоновку элементов меню или панели действий. Использование данных файлов освещается далее в п. 5.5.
- **Drawable** — файлы изображений, используемых приложением. Сюда также входят графические файлы, используемые в пользовательском интерфейсе.
- **Values** — данные приложения, представленные в текстовом формате XML. В первую очередь в состав ресурсов данного типа вхо-

дят все текстовые строки приложения, традиционно размещаемые в файле **strings.xml**. Пример такого файла приведён ниже:

```
<?xml version="1.0" encoding="utf—8"?>
<resources>
    <string name="app_name">Alarm clock</string>
    <string name="hours_label">Hours</string>
    <string name="minutes_label">Minutes</string>
</resources>
```

- **Raw** — произвольные данные, как правило бинарные.

Ресурсы каждого типа размещаются внутри каталога res проекта в подкаталогах, названия которых совпадают с типами ресурсов, приведёнными выше, но написаны строчными буквами.

5.3. Использование ресурсов из приложения. Для использования ресурсов из приложения в ходе сборки проекта (а при использовании инструментальных сред также при любом изменении ресурсов приложения) создаётся специальный файл R.java, содержащий идентификаторы всех ресурсов проекта. Все они определены как статические константы внутри подклассов класса **R**, каждый из которых соответствует своему типу ресурсов:

- **R.layout** — ресурсы из каталога res/layout, описывающие компоновку пользовательского интерфейса;
- **R.menu** — ресурсы из каталога res/menu, описывающие состав меню или панели действий;
- **R.id** — компоненты пользовательского интерфейса, описанные в файлах ресурсов (файлы каталога res/layout); элементы идентифицируются по идентификаторам, задаваемым с помощью атрибута **android:id**;
- **R.drawable** — ресурсы-изображения из каталога res/drawables;
- **R.string**, **R.integer**, **R.boolean**, **R.color**, **R.array** и т. д. — ресурсы из файлов данных (файлы каталога res/values), сгруппированные по типам этих данных;
- **R.raw** — прочие ресурсы из каталога res/raw.

Приведём примеры использования ресурсов из кода. Загрузка ресурса, описывающего пользовательский интерфейс приложения, из файла res/layout/main.xml осуществляется с помощью вызова:

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

Чтение текстового ресурса, определённого в файле res/strings.xml как

```
<?xml version="1.0" encoding="utf−8"?>
<resources>
    <string name="error_message">Error!</string>
</resources>
```

и вывод информации, заключённой в этом ресурсе, в виде всплывающего сообщения можно выполнить следующим образом:

```
Toast.makeText(this, R.string.error_message, Toast.LENGTH_LONG).show();
```

Перечисленные выше примеры использовали лишь идентификаторы ресурсов, которые передавались различным виджетам, а последние в свою очередь загружали необходимые ресурсы. Но существует возможность непосредственной загрузки с помощью методов класса **Resources**. Объект этого класса можно получить в результате вызова **getResources()** класса **Context**:

```
Resources resources = getResources();
```

Затем можно получить текстовый ресурс и ресурс-изображение следующим образом:

```
String text = resources.getText(R.string.app_name);
Drawable icon = resources.getDrawable(R.drawable.ic_launcher);
```

Также существует возможность ссылки на ресурсы из описания других ресурсов. Она используется для определения идентификаторов элементов в файлах описания пользовательского интерфейса:

```
<TextView android:id="@+id/counterTextView" />
```

подстановки строк из файлов res/values в качестве надписей виджетов:

```
<TextView android:text="@string/counterText" />
```

а также в файле манифеста:

```
<application android:label="@string/app_name"
    android:icon="@drawable/ic_launcher">
```

В последнем случае указываются текстовый ресурс, содержащий название приложения, и ресурс-изображение, являющееся его иконкой.

5.4. Ресурсы, зависящие от конфигурации. Как уже упоминалось, одной из целей использования ресурсов является поддержка их вариативности в зависимости от конфигурации системы. Наиболее важными элементами конфигурации, требующими такой вариативности, являются:

- разрешение экрана (в приложении необходимо использовать изображения, подходящие для установленного разрешения экрана пользователя во избежание снижения их качества при масштабировании);
- ориентация экрана (для книжной и альбомной ориентации, как правило, требуется различная компоновка элементов пользовательского интерфейса);
- текущая локаль (интерфейс пользователя приложения должен быть на том языке, который установлен в системе).

Для решения задачи вариативности файлы ресурсов, соответствующие различным конфигурациям, размещаются в различных каталогах. При этом суффиксы указывают, для какой конфигурации предназначены ресурсы из соответствующего каталога:

- layout — каталог описания интерфейса для книжной ориентации; layout-land — для альбомной;
- drawable — набор изображений по умолчанию; drawable-ldpi, drawable-mdpi, drawable-hdpi — наборы изображений для устройств с низким, средним и высоким разрешением соответственно;
- values — строки приложения для языка по умолчанию (английский), values-ru — строки приложения в русской локализации, values-de — строки приложения в немецкой локализации и т. д.

Приведём пример использования данного механизма для интернационализации приложения. Пусть файл ресурсов, приведённый в п. 5.2, имеет имя strings.xml и находится внутри каталога res/values проекта. Тогда для того, чтобы обеспечить интернационализацию приложения для русского языка, необходимо создать следующий файл

```
<?xml version="1.0" encoding="utf−8"?>
<resources>
    <string name="app_name">Будильник</string>
    <string name="hours_label">Часы</string>
    <string name="minutes_label">Минуты</string>
</resources>
```

и разместить его под именем strings.xml внутри каталога res/values-ru.

5.5. Использование ресурсов для формирования меню и панели действий. Рассмотрим ещё один пример использования ресурсов,

демонстрирующий формирование главного меню. В ранних версиях Android главное меню приложения вызывается нажатием на аппаратную кнопку «Menu». Начиная с Android 3 используется другой подход: элементы меню выстраиваются в правом углу строки заголовка приложения, которая в этом случае называется панелью действий (action bar). Принцип реализации одинаков для обоих механизмов. Приведём его описание.

Сначала меню необходимо описать в файле ресурсов. В данном случае будем считать, что такой файл имеет имя res/menu/main.xml:

```
<?xml version="1.0" encoding="utf−8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menu_save"
          android:icon="@drawable/ic_menu_save"
          android:title="@string/menu_save"
          android:showAsAction="ifRoom|withText" />
    <item android:id="@+id/menu_delete"
          android:icon="@drawable/ic_menu_delete"
          android:title="@string/menu_delete"
          android:showAsAction="ifRoom|withText" />
    <item android:id="@+id/menu_options"
          android:title="@string/menu_options"
          android:showAsAction="never" />
</menu>
```

В состав меню входят три пункта. Первый и второй оформлены как элементы панели действий (атрибут **android:showAsAction** присутствует и не равно «never»), для каждого из них определены значок и текст (атрибуты **android:icon** и **android:title** соответственно). Значение «ifRoom|withText» означает, что в панели действий должны отображаться как значок действия, так и его текстовое описание (последнее только при наличии достаточного количества свободного пространства в панели). Другие возможные значения включают «always», обозначающее, что элемент всегда должен отображаться, и «never», обозначающий, что элемент не будет отображаться в панели действий. Это используется в описании третьего элемента в приведённом выше примере. Пользователь сможет получить доступ к этому элементу только путём нажатия на кнопку «Menu». Скриншоты приложения с открытым меню в портретной и альбомной ориентации приведены на рис. 5.1.

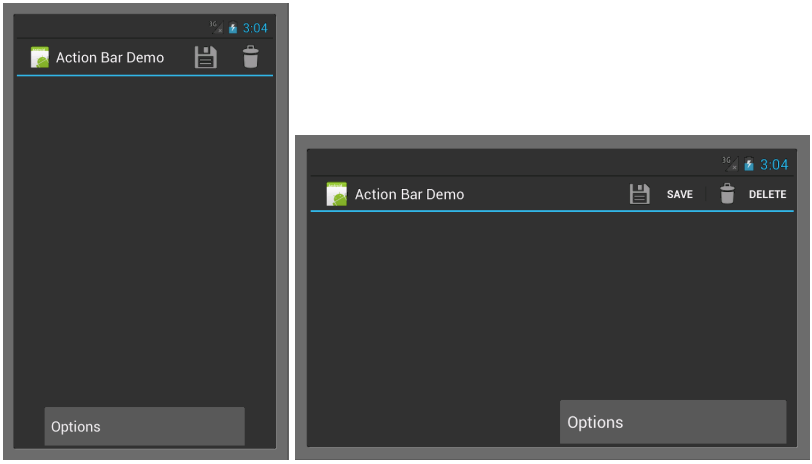

Рис. 5.1. Панель действий в портретной и альбомной ориентации

Чтобы загрузить приведённое описание в программном коде, необходимо переопределить метод **onCreateOptionsMenu()** класса активности:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.main, menu);
    return true;
}
```

В данный метод передаётся параметр **menu**, который необходимо использовать для формирования меню. В приведённом коде данная операция произведена с помощью класса **MenuInflater**, который способен загружать меню из XML-файлов описания. В метод **inflate()** передаётся идентификатор ресурса, из которого необходимо загрузить описание, а также объект-меню, подлежащий формированию.

5.6. Обработка действий меню и панели задач. Для обработки действий меню и панели задач необходимо переопределить метод **onOptionsItemSelected()** класса активности. Данный метод является диспетчером, поэтому типичный способ его реализации заключается в определении действия, которое было выбрано, с последующей обработкой этого действия:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
  case R.id.menu_save:
    // Handle the "save" operation
    break;
  case R.id.menu_delete:
    // Handle the "delete" operation
    break;
    // ...
  }
  return true;
}
```

5.7. Вопросы и упражнения для самопроверки:

1. Что такое ресурсы? Для решения каких задач разработан данный механизм в Android? Какие преимущества даёт разработчику использование механизма ресурсов?
2. Какие типы ресурсов существуют? Как размещены в проекте файлы ресурсов?
3. Как можно использовать ресурсы в приложении непосредственно из программного кода, а также из других ресурсов?
4. Что такое ресурсы, зависящие от конфигурации? Для чего предназначен данных механизм и как его можно использовать?
5. Как, используя механизм ресурсов, создать главное меню или панель действий Android-приложения?
6. В чём отличия в реализации меню для ранних и поздних версий платформы Android?
7. Как обработать выбор действий из главного меню или панели действий?

6. Хранение данных

6.1. Способы хранения данных. Многим приложениям требуется сохранять данные в постоянной памяти и восстанавливать их при последующих запусках. Платформа Android предоставляет три возможности решения данной задачи: настройки (prefereneces), файловая система и базы данных. Рассмотрим эти возможности более подробно.

6.2. Механизм настроек. Как следует из названия, основное назначение данного механизма состоит в хранении настроек приложения. Android API позволяет хранить настройки в виде пар «ключ—значение» и автоматически решает все задачи создания и управления файлами, в которых эти настройки хранятся.

Для того чтобы начать работать с настройками, необходимо получить объект класса **SharedPreferences** с помощью вызова метода

```
public SharedPreferences getSharedPreferences(String name, int mode);
```

класса **Context**. В качестве первого аргумента передаётся идентификатор файла настроек[4]. Второй аргумент определяет режим доступа. В большинстве случаев достаточно использовать режим по умолчанию (**Context.MODE_PRIVATE**). Другие значения данного параметра позволяют создавать настройки, разделяемые несколькими приложениями. О них можно подробнее узнать из документации.

После того как объект класса **SharedPreferences** получен, можно извлекать записанные ранее настройки с помощью методов[5]:

```
public String getString(String key, String defaultValue);
public int getInt(String key, int defaultValue);
...
```

Каждый из извлекаемых параметров идентифицируется по ключу (параметр **key**). Если запрошенного значения в файле настроек не оказы-

[4] Идентификатор может быть любой строкой, однако необходимо принимать меры по обеспечению уникальности этой строки. Например, для настроек, касающихся одной конкретной активности, разумно использовать в качестве идентификатора полное имя класса данной активности.

[5] В классе определены методы для каждого из примитивных типов и типа **String**.

вается, то возвращается значение по умолчанию — оно передаётся в вызов метода **get…()** в качестве второго аргумента.

Сохранение настроек осуществляется чуть сложнее. Сначала необходимо получить объект класса, реализующего интерфейс **SharedPreferences.Editor**, с помощью вызова метода

```
public SharedPreferences.Editor edit();
```

класса **SharedPreferences**. Полученный объект имеет методы

```
public SharedPreferences.Editor putString(String key, String value);
public SharedPreferences.Editor putInt (String key, int value);
…
```

позволяющие сохранять настройки соответствующих типов. Если значение параметра, соответствующего ключу, по которому осуществляется запись, уже присутствовало в файле настроек, это значение перезаписывается. Немедленно по окончании записи данных необходимо вызвать метод

```
public boolean commit();
```

который атомарно сохранит изменения в файле настроек.

Проиллюстрируем применение механизма настроек на следующем примере. Рассмотрим приложение «Счётчик» из главы 3 и добавим в него возможность сохранять состояние счётчика между запусками приложения. Несмотря на то, что состояние не является, вообще говоря, настройкой приложения, использование данного механизма в данном случае является самым простым решением задачи и потому представляется вполне уместным.

Будем осуществлять сохранение состояния в методе **onPause()**, а восстановление — в методе **onResume()**. Для того чтобы обеспечить восстановление состояния, определим метод установки значения в классе **Counter**:

```
public void setValue(int value) {
  this.value = value;
  if (listener != null) { listener.onModification(this); }
}
```

Из класса **MainActivity** необходимо удалить внесённый в п. 3.11 код сохранения и восстановления состояния счётчика при повороте, поскольку решение этой задачи в новой версии приложения будет

достигнуто автоматически. Полный код модифицированного класса **MainActivity**:

```
public class MainActivity extends Activity {
  private TextView counterText;
  private Counter counter = new Counter();

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    counterText = (TextView) findViewById(R.id.counterText);
    counter.setOnModificationListener(
        new Counter.OnModificationListener() {
      @Override
      public void onModification(Counter sender) { updateCounterView(); }
    });
  }

  public void updateCounterView() {
    counterText.setText(String.valueOf(counter.getValue()));
  }

  public void onIncreaseButtonClick(View v) {
    counter.increase();
  }

  public void onResetButtonClick(View v) {
    counter.reset();
  }

  @Override
  protected void onPause() {
    super.onPause();
    SharedPreferences prefs = getSharedPreferences(getLocalClassName(),
        Context.MODE_PRIVATE);
    SharedPreferences.Editor editor = prefs.edit();
    editor.putInt("counterValue", counter.getValue());
    editor.commit();
  }

  @Override
```

```
    protected void onResume() {
        super.onResume();
        SharedPreferences prefs = getSharedPreferences(getLocalClassName(),
                Context.MODE_PRIVATE);
        counter.setValue(prefs.getInt("counterValue", 0));
    }
}
```

6.3. Основные классы для работы СУБД SQLite. SQLite представляет собой встраиваемую СУБД, по умолчанию поддерживаемую в Android. Её использование в приложениях основывается на применении двух классов Android API: **SQLiteDatabase** и **SQLiteOpenHelper**. Первый инкапсулирует операции доступа к БД, включая добавление, изменение, удаление данных из таблиц, запросы на выборку данных, а также управление структурой БД. Второй класс является вспомогательным и предназначен для управления жизненным циклом БД, включая первоначальное создание схемы данных и обновление этой схемы при обновлении приложения.

6.4. Управление жизненным циклом БД. Разработчик Android-приложения сам ответствен за то, чтобы необходимая приложению база данных была создана перед началом использования и имела актуальную версию. Для того чтобы гарантировать это, следует создать класс, унаследованный от абстрактного класса **SQLiteOpenHelper**, определить конструктор и следующие методы:

```
public void onCreate(SQLiteDatabase db);
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion);
```

Конструктор унаследованного класса, как правило, просто вызывает конструктор суперкласса

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

с подходящими значениями параметров: в качестве **context** передаётся текущий контекст (можно просто передать ссылку на объект класса активности), **name** содержит имя базы данных, **version** — номер версии БД, а **factory** обычно устанавливается равным null.

Метод **onCreate()** вызывается в том случае, когда происходит самое первое с момента установки приложения обращение к БД. Типичная реализация данного метода заключается в выполнении SQL-команды **create**, создающей схему данных, посредством вызова метода

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

на объекте класса **SQLiteDatabase**.

Метод **onUpdate()** предназначен для внесения изменений в БД при обновлении приложения. Он вызывается в том случае, если номер текущей версии базы данных в системе не совпадает с числом, переданным в качестве параметра **version** в конструктор класса **SQLiteOpenHelper**. Номера старой и новой версий передаются в метод **onUpdate()** в качестве аргументов. На основании их сравнения программист может выполнить команды изменения схемы данных, преобразующие схему данных от старой версии к новой.

Как правило, объект класса, унаследованного от **SQLiteOpenHelper**, создаётся в методе **onCreate()** и помещается в поле класса активности. В дальнейшем на этом объекте вызывается метод

```
public SQLiteDatabase getWritableDatabase();
```

возвращающий объект класса **SQLiteDatabase**, через который осуществляется доступ к данным. По окончании использования необходимо закрыть БД посредством вызова метода

```
public void close();
```

на объекте класса **SQLiteOpenHelper**.

6.5. Доступ к данным. Класс **SQLiteDatabase** предоставляет множество методов для доступа к данным. Рассмотрим основные из них. Метод

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

предназначен для вставки строки в таблицу БД. Имя таблицы передаётся в качестве параметра **table**, а вставляемые значения — в качестве параметра **values**. В результата вызова метода возвращается количество добавленных строк или —1, если добавление осуществить не удалось.

Используемый для хранения значений параметров класс **ContentValues** представляет собой ассоциативный массив, в котором ключами являются имена столбцов таблицы, а значениями — данные соответствующих ячеек. Для записи пары «ключ—значение» в ассоциативный массив используется метод **put()** аналогично стандартным ассоциативным массивам Java.

Метод

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

предназначен для изменения записи в БД. Параметры **table** и **values** имеют такой же смысл, как и в случае метода **insert()**. Параметр **whereClause** содержит выражение на языке SQL, осуществляющее выборку записей для обновления. Данное выражение может включать подстановочные параметры, обозначаемые вопросительными знаками. Значения этих параметров передаются в виде массива **whereArgs** в том порядке, в котором они встречались в SQL-выражении.

Метод

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

предназначен для удаления записей из базы данных. Все параметры аналогичны уже рассмотренным.

Для получения данных из одной таблицы базы данных используется метод

```
public Cursor query(String table, String[] columns, String selection,
    String[] selectionArgs, String groupBy, String having,
    String orderBy, String limit);
```

Параметры метода означают следующее:

- **table** — имя таблицы, из которой осуществляется выборка;
- **columns** — список столбцов, которые надо вернуть в результате запроса (значение null означает все столбцы);
- **selection** — SQL-выражение для конструкции **where** SQL-запроса на выборку (может содержать подстановочные параметры);
- **selectionArgs** — значения подстановочных параметров;
- **groupBy**, **having**, **orderBy**, **limit** — SQL-выражения для конструкций **group by**, **having**, **order by** и **limit** запроса на выборку данных.

Большинство параметров являются необязательными и могут содержать значение null.

В случае, когда необходимо получить данные из нескольких таблиц, используется более общий метод

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

Данный метод принимает SQL-запрос, который непосредственно передаётся СУБД на исполнение.

6.6. Работа с курсорами. В результате выполнения методов **query()** и **rawQuery()** возвращается объект класса, реализующего интерфейс **Cursor**, который предназначен для навигации по результирующему набору данных.

Первоначально курсор находится в позиции, предшествующей первой строке набора данных. Для перехода к следующей позиции используется метод

```
public boolean moveToNext();
```

который возвращает true, если переход был успешным, и false, если набор данных закончился.

Для получения полей строки, через которую «перешагнул» курсор, используется один из методов

```
public int getInt(int columnIndex);
public long getLong(int columnIndex);
public String getString(int columnIndex);
...
```

в зависимости от типа получаемых данных. В качестве параметра **columnIndex** в приведённые выше методы передаётся порядковый номер столбца, значение которого требуется получить. При необходимости номер столбца результирующего набора данных, а также его тип можно получить по имени, используя методы:

```
public int getColumnIndex(String columnName);
public int getType(int columnIndex);
```

Допустимые типы данных определены как статические константы интерфейса **Cursor**.

Рассмотрим пример. Приведённая ниже функция выводит содержимое произвольной таблицы в файл журнала (это может быть полезно для отладки приложения). Для простоты предполагается, что столбцы таблицы могут иметь лишь целочисленный или строковый тип.

```
private void printTable(SQLiteDatabase database, String tableName) {
  Cursor cursor = database.query(tableName, null, null, null, null,
      null, null, null);
  while (cursor.moveToNext()) {
    Log.d(getLocalClassName(), "Record:");
```

```
        for (int i = 0; i < cursor.getColumnCount(); i++) {
            String columnName = cursor.getColumnName(i) + ": ";
            switch (cursor.getType(i)) {
            case Cursor.FIELD_TYPE_INTEGER:
                Log.d(getLocalClassName(), columnName + cursor.getInt(i));
                break;
            case Cursor.FIELD_TYPE_STRING:
                Log.d(getLocalClassName(), columnName + cursor.getString(i));
                break;
            }
        }
    }
}
```

6.7. Вопросы и упражнения для самопроверки:

1. Перечислите способы постоянного хранения данных на платформе Android. Объясните, в каких случаях разумно применять каждый из них.
2. Что такое механизм настроек? Для чего он предназначен? Как его применять?
3. Перечислите основные классы Android, предназначенные для работы с базой данных SQLite. На примерах объясните, как их применять.
4. Что такое жизненный цикл базы данных? Какие средства платформы Android позволяют управлять этим жизненным циклом?
5. Назовите методы класса **SQLiteDatabase**, предназначенные для работы с данными. На примерах объясните, как их можно использовать.
6. В чём отличие между методами **query()** и **rawQuery()** класса **SQLiteDatabase**? В каких случаях применяется каждый из них?
7. Что такое курсор набора данных? Для чего он предназначен? На примерах объясните, как использовать курсоры.

7. Пример приложения, использующего БД для хранения данных

7.1. Описание приложения. Данный раздел содержит пример приложения, использующего базу данных для хранения списка задачи пользователя (to-do list). Пример призван продемонстрировать, как может использоваться БД в приложении, включая выполнение таких действий, как управление жизненным циклом БД, добавление, изменение и получение записей из БД, отображение данных в списке и их редактирование в отдельной активности.

Приложение содержит две активности. Главная активность предназначена для отображения списка дел. Вторая активность позволяет редактировать атрибуты пользовательской задачи, включающие название, описание и дату выполнения. Внешний вид приложения представлен на рис. 7.1.

7.2. Класс управления жизненным циклом БД. Для управления жизненным циклом определим в приложении класс **DBHelper**, унаследованный от класса **SQLiteOpenHelper**:

```
public class DBHelper extends SQLiteOpenHelper {
    public DBHelper(Context context) {
        super(context, "todos", null, 1);
    }
    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("create table todos (" +
                "_id integer primary key autoincrement," +
                "title text," +
                "description text," +
                "dueDate text);"
        );
    }
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    }
}
```

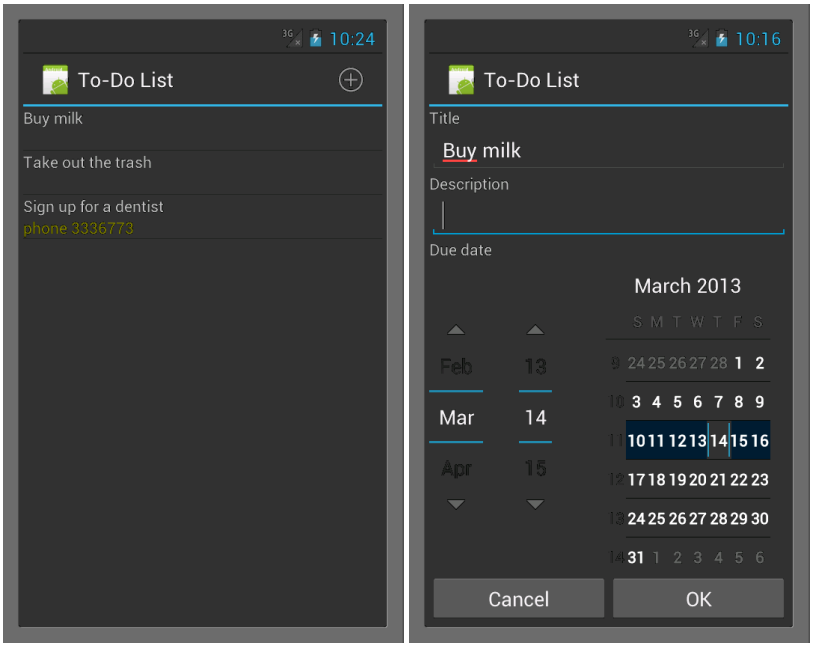

Рис. 7.1. Главная активность и активность редактора в приложении To-do List

Данный класс создаёт схему данных при первой попытке обращения к БД. Поскольку приложение имеет единственную версию, номер версии в конструкторе класса **SQLiteOpenHelper** установлен равным 1, а метод **onUpgrade()** оставлен пустым.

Каждая задача имеет название (title), описание (description) и дату выполнения (dueDate). Поскольку в SQLite нет специального типа данных для хранения дат, используется строковое представление в формате ISO 8601 (например, 2013-01-21). Для первичного ключа создано поле с названием _ id. Такое название необходимо для правильной работы адаптера списка, связанного с таблицей БД.

7.3. Пользовательский интерфейс главной активности. Описание интерфейса главной активности размещается в файле res/layout/main.xml:

```
<?xml version="1.0" encoding="utf—8"?>
<ListView android:id="@+id/todoList"
```

```
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    xmlns:android="http://schemas.android.com/apk/res/android">
</ListView>
```

и содержит единственный элемент — список с идентификатором **todoList**.

7.4. **Инициализация главной активности.** Главная активность размещается в классе **MainActivity**. Будем разбирать содержимое этого класса отдельными фрагментами по мере добавления функциональности в приложение. Начнём с кода инициализации главной активности:

```
public class MainActivity extends Activity {
  private DBHelper dbHelper;
  private Cursor cursor;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    ListView todoListView = (ListView) findViewById(R.id.todoList);
    todoListView.setOnItemClickListener(new ListView.OnItemClickListener() {
      @Override
      public void onItemClick(AdapterView<?> parent, View view,
          int position, long id) {
        onToDoListItemClick(id);
      }
    });
    dbHelper = new DBHelper(this);
    cursor = dbHelper.getWritableDatabase().query("todos", null, null, null,
        null, null, "dueDate");
    String[] from = new String[] { "title", "description" };
    int[] to = new int[] { R.id.titleText, R.id.descriptionText };
    SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
        R.layout.todo_item, cursor, from, to,
        CursorAdapter.FLAG_AUTO_REQUERY);
    todoListView.setAdapter(adapter);
  }
// ...
```

При инициализации активности происходит загрузка описания интерфейса из XML-файла (см. п. 7.3), к списку прикрепляется обработчик события выбора элемента, создаётся экземпляр класса **DBHelper** и помещается в поле класса. Далее осуществляется выборка всех задач из таблицы todos базы данных с сортировкой по сроку завершения в хронологическом порядке. В результате выполнения команды на выборку возвращается курсор, который также сохраняется в поле класса **MainActivity**. Далее создаётся экземпляр класса **SimpleCursorAdapter**, который осуществляет отображение записей полученного набора данных на текстовые поля элемента списка.

Внешний вид элемента списка описывается файлом res/layout/todo_item.xml. Каждый элемент содержит поле для названия задачи и поле для её описания.

```
<?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/titleText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="left|center_vertical"/>
  <TextView android:id="@+id/descriptionText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textColor="#808000"
        android:layout_gravity="left|center_vertical"/>
</LinearLayout>
```

7.5. Меню приложения и обработка добавления записи. Меню приложения (в формате панели действий) описывается в файле **res/menu/main.xml** и содержит кнопку добавления задачи:

```
<?xml version="1.0" encoding="utf—8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/add_todo"
            android:showAsAction="always"
            android:icon="@android:drawable/ic_menu_add" />
</menu>
```

Инициализация меню и обработка касания кнопки добавления записи осуществляются стандартным образом в классе **MainActivity**:

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

@Override
  public boolean onOptionsItemSelected(MenuItem item) {
    Intent intent = new Intent(this, ToDoEditorActivity.class);
    startActivityForResult(intent, 1);
    return true;
  }
// ...
```

Касание кнопки приводит к вызову активности **ToDoEditorActivity**, отвечающей за редактирование записи. Поскольку кнопка в панели действий является единственной, метод **onOptionsItemSelected()** не содержит кода проверки выбранного элемента меню.

7.6. Пользовательский интерфейс активности редактора. Интерфейс активности, предназначенной для редактирования атрибутов задач, включает в себя текстовые поля для ввода названия задачи и её описания, компонент **DatePicker** для ввода даты завершения задачи, две кнопки «OK» и «Cancel», а также метки (**TextView**) для описания назначения всех элементов интерфейса.

Для описания пользовательского интерфейса используется файл res/layout/todo_editor.xml. Он выглядит следующим образом:

```
<?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:text="Title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="left|center_vertical"/>
  <EditText android:id="@+id/titleText"
      android:layout_width="fill_parent"
```

```
        android:layout_height="wrap_content"
        android:layout_gravity="left|center_vertical"/>
    <TextView android:text="Description"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="left|center_vertical"/>
    <EditText android:id="@+id/descriptionText"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="left|center_vertical"/>
    <TextView android:id="@+id/textView" android:text="Due date"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="left|center_vertical"/>
    <DatePicker android:id="@+id/dueDatePicker"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />
    <LinearLayout android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
      <Button android:id="@+id/cancelButton" android:text="Cancel"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:layout_weight="1"
          android:layout_gravity="center_horizontal|bottom"
          android:onClick="onCancelButtonClick" />
      <Button android:id="@+id/okButton" android:text="OK"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:layout_weight="1"
          android:layout_gravity="center_horizontal|bottom"
          android:onClick="onOkButtonClick" />
    </LinearLayout>
  </LinearLayout>
```

7.7. **Интерфейс взаимодействия активностей.** Перед тем как перейти к рассмотрению реализации активности редактора, необходимо определить интерфейс взаимодействия этой активности с главной активностью приложения. Возможны два случая использования активности редактора: для создания новой записи или редактирования существующей.

Договоримся, что при создании новой записи никаких данных из главной активности не передаётся, тогда как при редактировании через дополнительные поля (extras) интента в активность редактора передаётся следующая информация:

- **int** id — идентификатор редактируемой записи в БД;
- **String** title — название редактируемой задачи;
- **String** description — описание редактируемой задачи;
- **String** dueDate — дата завершения редактируемой задачи в формате ISO 8601.

Активность редактора возвращает **RESULT_OK**, если пользователь принял внесённые изменения нажатием кнопки «OK», и **RESULT_CANCELED**, если пользователь отказался от изменений с помощью кнопки «Cancel» или аппаратной кнопки «Back» устройства.

Если пользователь принял изменения, то через механизм дополнительных полей интента возвращаются следующие значения:

- **int** id — идентификатор редактируемой записи в БД (только в том случае, если активность была вызвана для редактирования задачи; в противном случае данное значение не передаётся);
- **String** title — название добавляемой/отредактированной задачи;
- **String** description — описание добавляемой/отредактированной задачи;
- **String** dueDate — дата завершения добавляемой/отредактированной задачи в формате ISO 8601.

Отметим, что документирование способа взаимодействия чрезвычайно важно для разработки приложений. При отсутствии стандартизованного механизма описания интерфейсов в Android текстовое описание, подобное приведённому выше, может вполне успешно решать задачу передачи информации о правильном использовании активностей между разработчиками.

7.8. Реализация активности редактора задач. Реализация активности редактора размещается в классе **ToDoEditorActivity**:

```
public class ToDoEditorActivity extends Activity {
  private EditText titleText, descriptionText;
  private DatePicker dueDatePicker;
  private Intent resultIntent = new Intent();
  private static final SimpleDateFormat dateFormat
      = new SimpleDateFormat("yyyy-MM-dd");
```

```
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.todo_editor);
  titleText = (EditText) findViewById(R.id.titleText);
  descriptionText = (EditText) findViewById(R.id.descriptionText);
  dueDatePicker = (DatePicker) findViewById(R.id.dueDatePicker);
  if (getIntent().hasExtra("id")) {
    resultIntent.putExtra("id", getIntent().getIntExtra("id", 0));
    titleText.setText(getIntent().getStringExtra("title"));
    descriptionText.setText(getIntent().getStringExtra("description"));
    GregorianCalendar calendar = stringToDate(
        getIntent().getStringExtra("dueDate"));
    dueDatePicker.init(calendar.get(Calendar.YEAR),
        calendar.get(Calendar.MONTH),
        calendar.get(Calendar.DAY_OF_MONTH), null);
  }
}

private static String dateToString(int year, int month, int day) {
  GregorianCalendar calendar = new GregorianCalendar(year, month, day);
  return dateFormat.format(calendar.getTime());
}

private static GregorianCalendar stringToDate(String dateString) {
  try {
    Date date = dateFormat.parse(dateString);
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(date);
    return calendar;
  } catch (ParseException e) {
    return null;
  }
}

public void onOkButtonClick(View v) {
  resultIntent.putExtra("title", titleText.getText().toString());
  resultIntent.putExtra("description", descriptionText.getText().toString());
  resultIntent.putExtra("dueDate", dateToString(dueDatePicker.getYear(),
      dueDatePicker.getMonth(), dueDatePicker.getDayOfMonth()));
  setResult(RESULT_OK, resultIntent);
  finish();
```

```
    }

    public void onCancelButtonClick(View v) {
        setResult(RESULT_CANCELED);
        finish();
    }
}
```

Метод **onCreate()** инициализирует интерфейс активности редактора, сохраняет ссылки на необходимые компоненты интерфейса в полях класса активности, а также переносит значения из интента в соответствующие виджеты, если активность была открыта в режиме редактирования существующей записи (это определяется по наличию параметра id в дополнительных параметрах интента). Обработчик кнопки «OK» выполняет обратную передачу данных из виджетов в возвращаемый интент. Кроме того, этот обработчик, как и обработчик кнопки «Cancel», устанавливает возвращаемое значение и завершает выполнение активности с помощью вызова **finish()**.

Для выделения отдельных компонентов даты из строк в формате ISO 8601 и формирования таких строк из отдельных компонентов определены два вспомогательных метода: **dateToString()** и **stringToDate()** соответственно.

7.9. Вызов активности редактора для изменения существующей задачи. Вызов активности редактора в режиме добавления задачи уже рассмотрен в п. 7.5. Второй случай вызова осуществляется в том случае, когда пользователь касается одного из пунктов списка задач, вызывая соответствующую задачу на редактирование. Обработка данного действия осуществляется в методе **onToDoListItemClick()** главной активности:

```
// ...
    public void onToDoListItemClick(long id) {
        Cursor todoCursor = dbHelper.getReadableDatabase().query("todos", null,
                "_id = ?", new String[] { String.valueOf(id) }, null, null, null);
        todoCursor.moveToNext();
        Intent intent = new Intent(this, ToDoEditorActivity.class);
        intent.putExtra("id", todoCursor.getInt(
                todoCursor.getColumnIndex("_id")));
        intent.putExtra("title", todoCursor.getString(
                todoCursor.getColumnIndex("title")));
```

```
        intent.putExtra("description", todoCursor.getString(
                todoCursor.getColumnIndex("description")));
        intent.putExtra("dueDate", todoCursor.getString(
                todoCursor.getColumnIndex("dueDate")));
        startActivityForResult(intent, 1);
    }
// ...
```

Данный обработчик получает содержимое записи из БД по идентификатору, а затем заполняет дополнительные параметры интента значениями полей полученной записи, после чего вызывает активность редактора.

7.10. Обработка результата вызова активности редактора в главной активности. Наконец рассмотрим обработку возвращённого значения из активности редактора. Эта обработка осуществляется в методе **onActivityResult()** класса главной активности:

```
// ...
    @Override
    protected void onActivityResult(int requestCode, int resultCode,
    Intent data) {
        if (resultCode != RESULT_OK) return;
        ContentValues cv = new ContentValues();
        cv.put("title", data.getStringExtra("title"));
        cv.put("description", data.getStringExtra("description"));
        cv.put("dueDate", data.getStringExtra("dueDate"));
        if (data.hasExtra("id")) {
            dbHelper.getWritableDatabase().update("todos", cv, "_id = ?",
                    new String[] { String.valueOf(data.getIntExtra("id", 0)) });
        } else {
            dbHelper.getWritableDatabase().insert("todos", null, cv);
        }
        cursor.requery();
    }
} // class MainActivity
```

В том случае, когда пользователь не принял внесённых изменений, происходит возврат из обработчика. В противном случае атрибуты модифицированной задачи переносятся из возвращённого интента в объект класса **ContentValues**. Далее по наличию значения id в дополнительных параметрах интента выясняется действие, которое необходимо выполнить: добавление новой записи или изменение существу-

ющей. В результате вызывается метод **insert()** или метод **update()** на объекте базы данных. В последней строке метода вызывается метод **requery()** на курсоре, что вызывает обновление содержимого списка задач на экране.

7.11. Вопросы и упражнения для самопроверки:

1. Создайте проект приложения, описанный в этой главе. Скомпилируйте приложение и запустите его в эмуляторе или на реальном устройстве.
2. Добавьте в проект приложения возможность удаления записей.
3. Модифицируйте внешний вид элемента списка таким образом, чтобы в нём отображалось количество дней до завершения задачи вместо её описания.

   Указание. Используйте метод **rawQuery()** и функции для работы с датами SQLite для получения количества дней и его представления в удобном для пользователя виде (например, «due in 1 day», «due in 2 days», «overdue in 4 days» и т. д.).
4. Добавьте в базу данных дополнительное поле, содержащее дату добавления задачи. Используйте метод **onUpgrade()** и утверждение **alter table** языка SQL для корректного обновления базы данных первой версии.

8. Асинхронное выполнение

8.1. Назначение механизмов асинхронного выполнения. Асинхронное выполнение подразумевает использование отдельных потоков для выполнения некоторых действий. Необходимость в асинхронном выполнении чаще всего вызвана наличием в приложении некоторых процессов (вычисления, обращение к сетевым ресурсам, чтение из базы данных), требующих достаточно высоких временных затрат. Если не переносить выполнение данных процессов на отдельные потоки, это неминуемо скажется на отзывчивости интерфейса пользователя, снижая удобство приложения.

Организация асинхронного выполнения может опираться на стандартный механизм потоков Java (класс **Thread** и интерфейс **Runnable**) либо использовать Android-специфичные API, являющиеся высокоуровневыми обёртками стандартных потоков.

Важной проблемой асинхронного выполнения является синхронизация потоков. Необходимость её связана с тем фактом, что средства Android API, как и любой библиотеки графического интерфейса, являются поточно небезопасными. Последнее означает, что обращение к виджетам с потоков, отличных от главного потока выполнения приложения, может приводить к непредсказуемым последствиям.

Для решения проблемы синхронизации Android API предоставляет различные средства, включающие очереди сообщений, класс для организации выполнения асинхронных задач **AsyncTask**, а также более специализированные механизмы, ориентированные на решение отдельных задач, таких как асинхронная загрузка данных из БД.

8.2. Класс Handler и очередь сообщений. Класс **Handler** предназначен для управления очередями сообщений, связанными с потоками выполнения. Сообщения могут отправляться в очередь с любого потока выполнения, но обрабатываются они всегда на главном (связанном с пользовательским интерфейсом) потоке. Таким образом класс **Handler** обеспечивает синхронизацию потоков.

Для отправки сообщения в очередь используются методы

```
public boolean sendMessage(Message msg);
```

```
public boolean sendMessageDelayed(Message msg, long delayMillis);
public boolean sendMessageAtTime(Message msg, long uptimeMillis);
```

Сообщение представляет собой объект класса **Message**. Для хранения деталей передаваемого сообщения могут использоваться целочисленные свойства с именами **what**, **arg1** и **arg2**, а также свойство **obj** типа **Object**. Android API не регламентирует способ применения данных свойств, поэтому разработчик может использовать их по собственному усмотрению.

Для оптимизации использования оперативной памяти рекомендуется не создавать объекты класса **Message** с помощью операции **new**, а вызывать один из статических методов класса **Handler**:

```
public Message obtainMessage(int what, int arg1, int arg2, Object obj);
public Message obtainMessage(int what);
public Message obtainMessage(int what, Object obj);
...
```

Данные методы обеспечивают повторное использование объектов сообщений, организуя пул. В случае запроса объекта с повторяющимися значениями параметров эти методы просто возвращают его из пула, а не создают новый объект.

Для обработки сообщений необходимо унаследовать собственный класс от класса **Handler**, переопределить метод

```
public void handleMessage(Message msg);
```

и в нём разместить код обработки сообщения, которое передаётся в метод в качестве аргумента. Обработка сообщения осуществляется на главном потоке выполнения приложения, поэтому из данного обработчика можно обращаться к элементам пользовательского интерфейса.

Помимо отправки сообщений в очередь можно добавлять объекты классов, реализующих интерфейс **Runnable**. Это осуществляется с помощью методов

```
public boolean post(Runnable r);
public boolean postDelayed(Runnable r, long delayMillis);
public boolean postAtTime (Runnable r, long uptimeMillis);
```

При использовании данного подхода в роли обработчика выступает метод **run()** передаваемого объекта.

8.3. Пример использования класса Handler. В качестве примера использования класса **Handler** рассмотрим приложение, определяющее внешний IP-адрес устройства с помощью онлайн-сервиса Google. Поскольку выполнение сетевого запроса требует времени, крайне нежелательно выполнять подобные действия на основном потоке выполнения. Для решения данной проблемы создадим отдельный поток, который будет выполнять запрос, а затем передавать полученный в результате запроса к сервису IP-адрес на главный поток приложения.

Интерфейс приложения содержит кнопку для инициирования запроса, индикатор прогресса и поле для вывода результата. Файл res/layout/main.xml, описывающий пользовательский интерфейс главной активности, выглядит следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
  <LinearLayout android:orientation="horizontal"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="left|center_vertical">
    <Button android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Determine IP address"
        android:onClick="onDetermineIPAddressClick"/>
    <ProgressBar android:id="@+id/progressBar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:visibility="invisible"/>
  </LinearLayout>
  <TextView android:id="@+id/ipTextView"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="left|center_vertical"/>
</LinearLayout>
```

Поскольку для правильного функционирования приложению требуется доступ в Интернет, необходимо добавить соответствующее разрешение в файл манифеста:

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

Инициализация класса главной активности стандартна, в методе **onCreate()** происходит заполнение полей класса активности, хранящих ссылки на виджеты пользовательского интерфейса:

```
public class MainActivity extends Activity {
    private Handler handler;
    private TextView ipTextView;
    private ProgressBar progressBar;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        ipTextView = (TextView) findViewById(R.id.ipTextView);
        progressBar = (ProgressBar) findViewById(R.id.progressBar);
        handler = new Handler() {
            @Override
            public void handleMessage(Message msg) {
                handleIPDeterminationMessage(msg);
            }
        };
    }
// ...
```

В поле **handler** записывается объект анонимного класса, унаследованного от **Handler**, который передаёт обработку сообщения методу **handleIPDeterminationMessage()** класса главной активности.

Обработчик кнопки «Determine IP address» выглядит следующим образом:

```
// ...
    public void onDetermineIPAddressClick(View v) {
        ipTextView.setText("");
        progressBar.setVisibility(View.VISIBLE);
        Thread determinationThread = new Thread() {
            @Override
            public void run() {
                determineIPAddress();
            }
        };
        determinationThread.start();
    }
// ...
```

Данный обработчик очищает поле вывода результата, делает видимым индикатор прогресса, а затем инициирует выполнение метода **determineIPAddress()** класса главной активности на отдельном потоке выполнения. Указанный метод осуществляет запрос к web-сервису, обрабатывает возвращённое значение в формате JSON и отправляет полученный IP-адрес на главный поток выполнения в виде сообщения:

```
// ...
    private void determineIPAddress() {
        try {
            URL url = new URL(
                    "http://ip2country.sourceforge.net/ip2c.php?format=JSON");
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            conn.connect();
            BufferedReader reader = new BufferedReader(
                    new InputStreamReader(conn.getInputStream()));
            String ip = (String) new JSONObject(reader.readLine()).get("ip");
            reader.close();
            handler.sendMessage(handler.obtainMessage(0, 0, 0, ip));
        } catch (Exception e) {
            handler.sendMessage(handler.obtainMessage(0, 0, 0, e.getMessage()));
        }
    }
// ...
```

Класс **Handler** обрабатывает сообщение, вызывая метод **handleIPDeterminationMessage()** и передавая ему сообщение в качестве аргумента:

```
// ...
    private void handleIPDeterminationMessage(Message msg) {
        progressBar.setVisibility(View.INVISIBLE);
        ipTextView.setText(msg.obj.toString());
    }
} // class MainActivity
```

Данный метод скрывает индикатор прогресса, извлекает определённый IP-адрес из сообщения и помещает его в текстовое поле.

8.4. Класс AsyncTask. Помимо использования класса **Handler**, Android API предоставляет более простой подход для организации асинхронного выполнения. Этот способ основан на применении клас-

са **AsyncTask**, инкапсулирующего некоторую задачу, имеющую входные и выходные параметры и требующую выполнения на отдельном потоке. При использовании данного класса нет необходимости создавать поток явно, достаточно лишь унаследовать от **AsyncTask** собственный класс и переопределить несколько методов.

Отметим, что **AsyncTask** — это параметризованный класс. Его параметры обозначаются **<Params**, **Progress**, **Result>** и представляют собой типы входных данных, промежуточного и окончательного результата выполняемой задачи. Основные переопределяемые методы класса включают в себя:

```
protected void onPreExecute();
protected Result doInBackground(Params... params);
protected void onProgressUpdate(Progress... values);
protected void onPostExecute(Result result);
```

Главный из перечисленных методов — **doInBackground()**. Он содержит собственно код задачи, выполняемой на отдельном потоке. Остальные приведённые методы вызываются на главном потоке выполнения и, следовательно, могут обращаться к элементам пользовательского интерфейса. Синхронизация между потоками осуществляется классом **AsyncTask** автоматически. Методы **onPreExecute()** и **onPostExecute()** выполняются соответственно перед запуском и после завершения асинхронной задачи. Выполнение метода **onProgressUpdate()** инициируется вызовом

```
protected void publishProgress(Progress... values);
```

из кода метода **doInBackground()**. Данная возможность используется для вывода пользователю промежуточных результатов или статуса во время выполнения задачи.

8.5. Пример использования класса AsyncTask. Проиллюстрируем приведённую в предыдущем пункте информацию, заменив класс **Handler** в примере из п. 8.3 классом **AsyncTask**. Сначала удалим поле типа **Handler** и его инициализацию из класса **MainActivity**:

```
public class MainActivity extends Activity {
  private TextView ipTextView;
  private ProgressBar progressBar;

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

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        ipTextView = (TextView) findViewById(R.id.ipTextView);
        progressBar = (ProgressBar) findViewById(R.id.progressBar);
    }
// ...
```

Класс, унаследованный от **AsyncTask**, назовём **IPDeterminationTask** и разместим внутри класса активности. Это необходимо для упрощения обновления содержимого виджетов, являющихся полями класса **Activity**. Код определения IP-адреса будет размещаться в методе **doInBackground()**:

```
// ...
    private class IPDeterminationTask extends AsyncTask<Void, Void, String> {
        @Override
        protected String doInBackground(Void... params) {
            try {
                URL url = new URL(
                        "http://ip2country.sourceforge.net/ip2c.php?format=JSON");
                HttpURLConnection conn =
                        (HttpURLConnection) url.openConnection();
                conn.connect();
                BufferedReader reader = new BufferedReader(
                        new InputStreamReader(conn.getInputStream()));
                String ip = (String) new JSONObject(reader.readLine()).get("ip");
                reader.close();
                return ip;
            } catch (Exception e) {
                return e.getMessage();
            }
        }
// ...
```

Возвращаемое значение имеет тип **String** и по окончании выполнения задачи содержит результат определения IP-адреса. Методы **onPreExecute()** и **onPostExecute()** используются для обновления значения текстового поля результата, а также для отображения и сокрытия индикатора прогресса. Заметим, что вывод результата упрощается, т. к. возвращаемое значение метода **doInBackground()** передаётся в качестве параметра в метод **onPostExecute()**:

```
// ...
```

```
        @Override
        protected void onPreExecute() {
            ipTextView.setText("");
            progressBar.setVisibility(View.VISIBLE);
        }

        @Override
        protected void onPostExecute(String ip) {
            progressBar.setVisibility(View.INVISIBLE);
            ipTextView.setText(ip);
        }
    } // class IPDeterminationTask
// ...
```

Наконец, функции обработчика кнопки «Determine IP address» сводятся к созданию экземпляра класса **IPDeterminationTask** и запуску выполнения задачи:

```
// ...
    public void onDetermineIPAddressClick(View v) {
        new IPDeterminationTask().execute();
    }
} // class MainActivity
```

8.6. Вопросы и упражнения для самопроверки:

1. Что такое асинхронное выполнение? В каких случаях оно используется? Какие задачи решает?
2. Перечислите средства асинхронного выполнения, предоставляемые Android API.
3. Что такое очередь сообщений? Какую функцию выполняет класс **Handler** и как его правильно использовать?
4. Для чего предназначен класс **AsyncTask**? Как его использовать?
5. Объясните, что такое синхронизация потоков. В каких случаях она необходима? Как средства платформы Android помогают решать задачу синхронизации?
6. Модифицируйте примеры из данной главы таким образом, чтобы асинхронное выполнение не прерывалось при изменении конфигурации приложения (например, при смене ориентации экрана).

9. Провайдеры контента

9.1. Назначение провайдеров контента. Провайдеры контента входят в число основных компонентов Android-приложений, наряду с активностями и сервисами. Их задача состоит в предоставлении данных множеству приложений без привязки к конкретному способу хранения этих данных.

Операции, предоставляемые провайдером контента, аналогичны типичным операциям работы с базой данных и включают в себя добавление, обновление и удаление записей, а также выполнение запросов на выборку данных. Следует отметить, что природа этих данных может быть произвольной, поскольку и их структура, и способ хранения всецело находятся в зоне ответственности провайдера контента.

Для взаимодействия с провайдером контента клиенты специфицируют URI данных, над которыми выполняется действие, в виде: *content://имя_провайдера_контента/спецификация_данных*. Имя провайдера контента должно совпадать со значением, которое указывается при регистрации провайдера контента в файле манифеста. Спецификация данных может иметь любой формат, однако общепринятой является REST-подобная спецификация.

9.2. Пример стандартного провайдера контента. В качестве примера рассмотрим стандартный провайдер контента, предоставляющий доступ к контактам пользователя мобильного устройства. Данный провайдер использует URI вида *content://com.android.contacts/contacts* для получения всех доступных контактов и URI *content://com.android.contacts/contacts/1* для доступа к данным контакта под номером 1. Причём последний URI можно использовать не только для получения данных контакта, но и для их изменения или удаления.

9.3. Провайдер контента для списка задач. Остаток главы посвящён рассмотрению механизма провайдеров контента на примере, в качестве которого будет выступать модифицированное приложение из главы 7. Здесь мы заменим непосредственное обращение к базе данных на взаимодействие с ней через провайдер контента. Код данного провайдера будет выглядеть следующим образом:

```
public class ToDoContentProvider extends ContentProvider {
  private DBHelper dbHelper;

  @Override
  public boolean onCreate() {
    dbHelper = new DBHelper(getContext());
    return false;
  }
// ...
```

В методе **onCreate()** создаётся экземпляр класса **DBHelper**, определённый в п. 7.2. Он используется для связи с хранилищем данных провайдера контента, в роли которого выступает база данных SQLite.

```
// ...
  private static final String AUTHORITY =
      "ru.ac.uniyar.todoslist.contentprovider";
  private static final String BASE_PATH = "todos";
  private static final int TODOS = 10;
  private static final int TODO_ID = 20;
  private static final UriMatcher matcher =
      new UriMatcher(UriMatcher.NO_MATCH);

  static {
    matcher.addURI(AUTHORITY, BASE_PATH, TODOS);
    matcher.addURI(AUTHORITY, BASE_PATH + "/#", TODO_ID);
  }

  @Override
  public Cursor query(Uri uri, String[] projection, String selection,
  String[] selectionArgs, String sortOrder) {
    SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
    queryBuilder.setTables("todos");
    int uriType = matcher.match(uri);
    switch (uriType) {
    case TODOS:
      break;
    case TODO_ID:
      queryBuilder.appendWhere("_id = " + uri.getLastPathSegment());
      break;
    default:
```

```
        throw new IllegalArgumentException("Unknown URI: " + uri);
    }
    SQLiteDatabase db = dbHelper.getWritableDatabase();
    Cursor cursor = queryBuilder.query(db, projection, selection,
            selectionArgs, null, null, sortOrder);
    cursor.setNotificationUri(getContext().getContentResolver(), uri);
    return cursor;
}

@Override
public String getType(Uri uri) {
    return null;
}

@Override
public Uri insert(Uri uri, ContentValues values) {
    int uriType = matcher.match(uri);
    SQLiteDatabase sqlDB = dbHelper.getWritableDatabase();
    long id = 0;
    switch (uriType) {
    case TODOS:
        id = sqlDB.insert("todos", null, values);
        break;
    default:
        throw new IllegalArgumentException("Unknown URI: " + uri);
    }
    getContext().getContentResolver().notifyChange(uri, null);
    return Uri.parse(BASE_PATH + "/" + id);
}

@Override
public int update(Uri uri, ContentValues values, String selection,
String[] selectionArgs) {
    int uriType = matcher.match(uri);
    SQLiteDatabase sqlDB = dbHelper.getWritableDatabase();
    int rowsUpdated;
    switch (uriType) {
    case TODOS:
        rowsUpdated = sqlDB.update("todos", values, selection, selectionArgs);
        break;
    case TODO_ID:
```

```
            String id = uri.getLastPathSegment();
            if (TextUtils.isEmpty(selection)) {
                rowsUpdated = sqlDB.update("todos", values, "_id =" + id, null);
            } else {
                rowsUpdated = sqlDB.update("todos", values, "_id =" + id +
                        " and " + selection, selectionArgs);
            }
            break;
        default:
            throw new IllegalArgumentException("Unknown URI: " + uri);
        }
        getContext().getContentResolver().notifyChange(uri, null);
        return rowsUpdated;
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        throw new UnsupportedOperationException();
    }
} // class ToDoContentProvider
```

Методы **query()**, **insert()**, **update()** и **delete()** являются переопределёнными методами класса **ContentProvider**. Каждый из этих методов принимает URI, специфицирующий данные, над которыми производится соответствующее действие. Методы **query()** и **update()** поддерживают URI вида *content://ru.ac.uniyar.todoslist.contentprovider/todos* и *content://ru.ac.uniyar.todoslist.contentprovider/todos/1* аналогично примеру с контактами пользователя, рассмотренному в п. 9.2. Метод **insert()** поддерживает лишь URI вида *content://ru.ac.uniyar.todoslist.contentprovider/todos*, поскольку в момент добавления никакого идентификатора записи ещё не присвоено. Метод **delete()** для данного провайдера контента не реализован (выбрасывает исключение), поскольку в рассматриваемом примере необходимости в нём нет.

Рассматривая реализацию перечисленных методов, можно отметить, что основная функция провайдера контента в данном случае сводится к преобразованию запросов к провайдеру в запросы к БД, хранящей данные о задачах.

9.4. Регистрация провайдера контента в файле манифеста. Каждый провайдер контента должен быть зарегистрирован в файле манифеста. В нашем примере регистрация выглядит следующим образом:

```
<provider
  android:name=".ToDoContentProvider"
  android:authorities="ru.ac.uniyar.todoslist.contentprovider" >
</provider>
```

После установки приложения доступ к провайдеру контента можно будет получить из любых приложений, используя соответствующие URI.

9.5. Асинхронная загрузка данных, предоставляемых провайдером контента. При использовании провайдеров контента существует очень полезная возможность асинхронной загрузки данных с помощью класса **CursorLoader**. Этот класс осуществляет запрос к провайдеру контента на отдельном потоке выполнения и вызывает callback-метод, когда загрузка завершается и можно использовать полученные данные.

Для использования класса **CursorLoader** необходим класс, реализующий интерфейс **LoaderManager.LoaderCallbacks**, который определяет все необходимые callback-методы. По соглашению таким классом обычно является класс активности:

```
public class MainActivity extends Activity
implements LoaderManager.LoaderCallbacks<Cursor> {
  private SimpleCursorAdapter adapter;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    ListView todoListView = (ListView) findViewById(R.id.todoList);
    todoListView.setOnItemClickListener(
        new ListView.OnItemClickListener() {
      @Override
      public void onItemClick(AdapterView<?> parent, View view,
      int position, long id) {
        onToDoListItemClick(id);
      }
    });
    getLoaderManager().initLoader(0, null, this);
    String[] from = new String[] { "title", "description" };
    int[] to = new int[] { R.id.titleText, R.id.descriptionText };
    adapter = new SimpleCursorAdapter(this, R.layout.todo_item, null,
```

```
            from, to, 0);
        todoListView.setAdapter(adapter);
    }
// ...
```

Заметим, что, в отличие от метода **onCreate()** из п. 7.4, в данном примере отсутствует запрос к БД, а объект класса **SimpleCursorAdapter** создаётся не связанным с конкретным курсором (третий параметр конструктора равен null). Это делается потому, что курсор будет создан динамически классом **CursorLoader** по окончании загрузки данных. Метод **initLoader()** инициирует запуск метода **onCreateLoader()** интерфейса **LoaderManager.LoaderCallbacks**:

```
// ...
    @Override
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
        return new CursorLoader(this,
            Uri.parse("content://ru.ac.uniyar.todoslist.contentprovider/todos"),
            null, null, null, null);
    }
// ...
```

По окончании загрузки данных вызывается callback-метод **onLoadFinished()**, которому передаётся курсор, связанный с полученными данными:

```
// ...
    @Override
    public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
        adapter.swapCursor(data);
    }
// ...
```

Метод **swapCursor()** приводит к отображению полученных данных в списке главной активности. Когда данные становятся недоступными, вызывается метод **onLoaderReset**:

```
// ...
    @Override
    public void onLoaderReset(Loader<Cursor> loader) {
        adapter.swapCursor(null);
    }
// ...
```

9.6. Вставка и обновление данных через провайдер контента. При использовании провайдера контента вставка и обновление данных происходят практически так же, как и в случае непосредственного выполнения данных операций с базой данных. Единственное отличие заключается в том, что методы получения и изменения данных вызываются не на объекте класса **SQLiteDatabase**, а на объекте класса **ContentResolver**, предоставляющего доступ к данным через провайдер контента, идентифицированного по URI:

```
// ...
    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
      if (resultCode != RESULT_OK) return;
      ContentValues cv = new ContentValues();
      cv.put("title", data.getStringExtra("title"));
      cv.put("description", data.getStringExtra("description"));
      cv.put("dueDate", data.getStringExtra("dueDate"));
      if (data.hasExtra("id")) {
        getContentResolver().update(
            Uri.parse("content://ru.ac.uniyar.todoslist.contentprovider/todos/" +
                data.getIntExtra("id", 0)), cv, null, null);
      } else {
        getContentResolver().insert(
            Uri.parse("content://ru.ac.uniyar.todoslist.contentprovider/todos/"), cv);
      }
    }

    public void onToDoListItemClick(long id) {
      Cursor todoCursor = getContentResolver().query(
          Uri.parse("content://ru.ac.uniyar.todoslist.contentprovider/todos/" + id),
          null, null, null, null);
      todoCursor.moveToNext();
      Intent intent = new Intent(this, ToDoEditorActivity.class);
      intent.putExtra("id", todoCursor.getInt(
          todoCursor.getColumnIndex("_id")));
      intent.putExtra("title", todoCursor.getString(
          todoCursor.getColumnIndex("title")));
      intent.putExtra("description", todoCursor.getString(
          todoCursor.getColumnIndex("description")));
      intent.putExtra("dueDate", todoCursor.getString(
          todoCursor.getColumnIndex("dueDate")));
```

```
        startActivityForResult(intent, 1);
    }
} // class MainActivity
```

Важная особенность класса **CursorLoader** заключается в том, что он автоматически отслеживает изменения в данных, получение которых он осуществляет. При изменении этих данных процесс загрузки производится заново без дополнительных усилий со стороны программиста. Поэтому при изменении данных нет необходимости в вызове метода **requery()**, который использовался в п. 7.10.

Заметим, что код методов работы с меню **onCreateOptionsMenu()** и **onOptionsItemSelected()** не меняется по сравнению с приведённым в п. 7.5.

После сборки и запуска приложения легко убедиться, что оно функционирует точно так же, как и приложение из главы 7.

9.7. Вопросы и упражнения для самопроверки:

1. Что такое провайдер контента? Какую роль играют провайдеры контента в инфраструктуре Android?
2. Для чего нужен URI при использовании провайдера контента? Из каких частей он состоит? По каким правилам формируется этот URI?
3. Какие методы класса **ContentProvider** необходимо переопределять при реализации провайдера контента?
4. Каково предназначение класса **CursorLoader**? Какие преимущества несёт в себе использование класса **CursorLoader** по сравнению с непосредственным выполнением запросов к провайдеру контента на главном потоке приложения?
5. Для чего предназначены callback-методы интерфейса **LoaderManager.LoaderCallbacks**? Приведите пример их использования.
6. Создайте проект приложения, описанный в этой главе. Скомпилируйте приложение и запустите его в эмуляторе или на реальном устройстве.
7. Добавьте возможность удаления записей в приложении-примере.
   Указание. Начните решение задачи с реализации метода **delete()** провайдера контента.

Литература

1. Программирование под Android / З. Медникс, Л. Дорнин, Б. Мик, М. Накамура. — СПб. : Питер, 2012. — 496 с.
2. Коматинени, С. Android 4 для профессионалов. Создание приложений для планшетных компьютеров и смартфонов / С. Коматинени, Д. Маклин. — М. : Вильямс, 2012. — 880 с.
3. Android для программистов. Создаем приложения / П. Дейтел, Х. Дейтел, Э. Дейтел, М. Моргано. — СПб. : Питер, 2012. — 560 с.
4. Левин, А. Android на планшетах и смартфонах / А. Левин. — СПб. : Питер, 2013. — 224 с.
5. Парамонов, И. В. Язык программирования Java и Java-технологии / И. В. Парамонов. — Ярославль : ЯрГУ, 2006. — 92 с.
6. Develop | Android Developers. — 2013. — URL: *http://developer.android.com/develop/index.html* (online; accessed: 01.02.2013).
7. Vogel, L. Android Development. Tutorials about development for Android. — 2013. — URL: *http://www.vogella.com/android.html* (online; accessed: 01.02.2013).
8. Nudelman, G. Android Design Patterns: Interaction Design Solutions for Developers / G. Nudelman. — Indianapolis : Wiley, 2013. — 458 p.
9. Friesen, J. Android Recipes: A Problem-solution Approach / J. Friesen, D. Smith. — N.-Y. : Apress, 2011.
10. Darwin, I. Android Cookbook / I. Darwin. — Sebastopol : O'Reilly Media Incorporated, 2012. — 688 p.
11. Haseman, C. Creating Android Applications: Develop and Design / C. Haseman. — Berkeley : Peachpit Press, 2011. — 273 p.
12. Ostrander, J. Android Ui Fundamentals: Develop & Design / J. Ostrander. — Berkeley : Peachpit Press, 2012. — 337 p.

Учебное издание

Парамонов Илья Вячеславович

Разработка мобильных приложений для платформы Android

Учебное пособие

Редактор, корректор М. Э. Левакова
Компьютерный набор, вёрстка И. В. Парамонова

Подписано в печать 07.05.2013 г. Формат 60×84/16.
Бумага тип. Усл. печ. л. 5,11. Уч.-изд. л. 5,0.
Тираж 100 экз. Заказ

Оригинал-макет подготовлен в редакционно-издательском отделе
Ярославского государственного университета.

Ярославский государственный университет им. П. Г. Демидова
150000, Ярославль, ул. Советская, 14.