Разработка мобильных приложений под Android

Курс I

Rev 1.11

Оглавление

Что такое Android?

История возникновения
Изначально разработкой ОС «*Android*» занималась компания *Android* Inc. В 2005 году Google выкупила *Android* за $130 млн. Как отметил Дэвид Лове, вице-президент компании, на симпозиуме в Стэнфорде: «Приобретение *Android* — это самое значительное приобретение Google за все время существования компании».
Однако днем рождения ОС принято считать другую дату — 05 ноября 2007г., когда Google объявила о создании Open Handset Alliance (OHA) — группы компаний, занимающихся разработкой стандартов ПО для мобильных устройств.
OHA представляет собой сообщество из более чем 50 компаний, включающее производителей программного и аппаратного обеспечения, а также мобильных операторов. Среди наиболее значительных членов Альянса можно назвать компании Motorola, HTC, Qualcomm, T-Mobile.

Архитектура Android

С точки зрения архитектуры *Android* является программным стеком для мобильных устройств, который включает операционную систему, связующее ПО и ключевые приложения.
Набор инструментов разработки под *Android* (*Android SDK*) содержит инструменты и интерфейсы прикладного программирования (API), требуемые для разработки приложений для платформы *Android* с использованием языка программирования Java.
С декабря 2014 года официальной средой разработки приложений для Android считается *Android Studio*, основанная на *IntelliJ IDEA*.

Особенности платформы Android

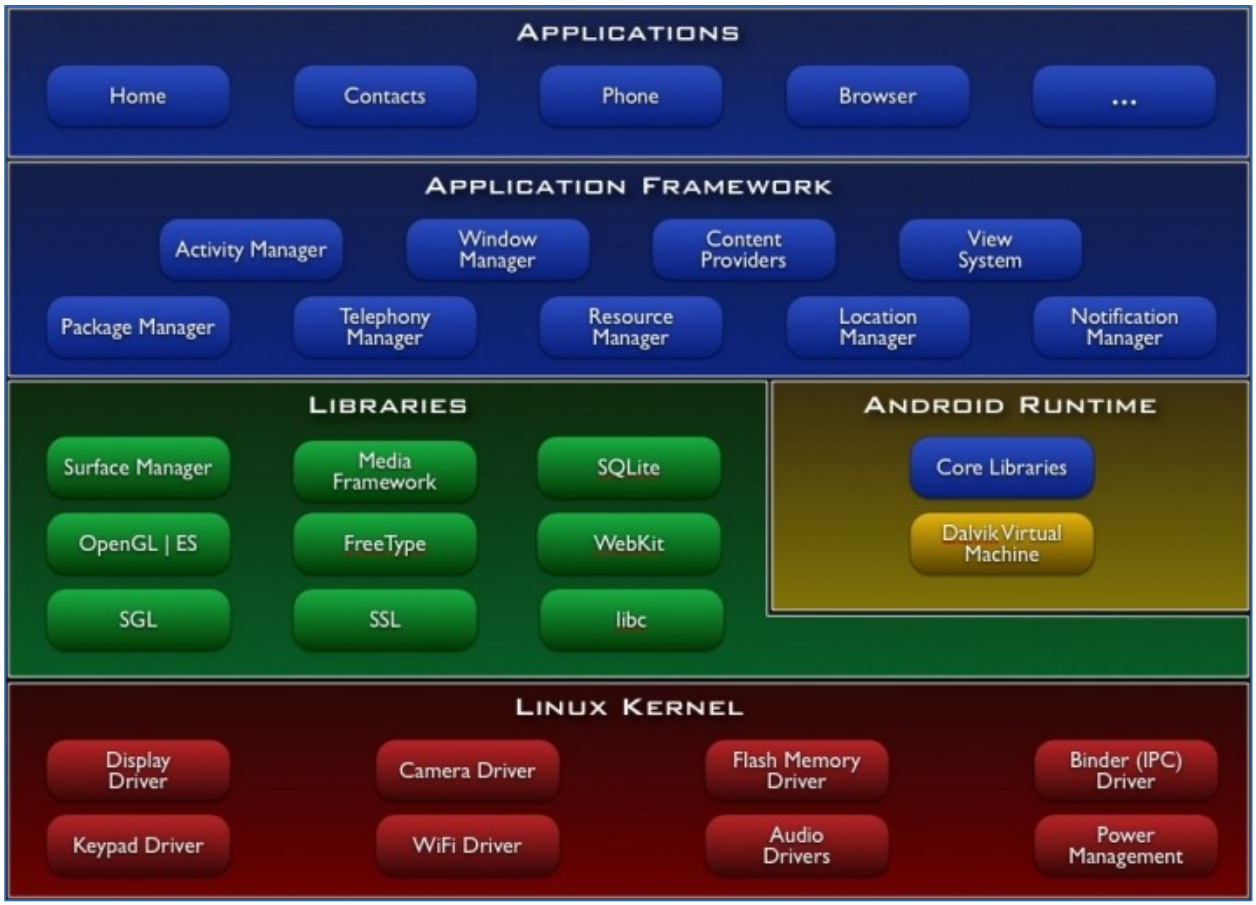
В состав *Android* входят разнообразные компоненты, интересные для разработчиков:

- Фреймворк приложений, позволяющий повторно использовать и взаимозаменять программные компоненты.
- Виртуальная машина *Dalvik*, оптимизированная для работы на мобильных устройствах, которая является промежуточным слоем. Начиная с *Android KitKat*, Google аннонсировала курс на замену *Dalvik* (использующей технологию комптляции JIT, Just-In-Time) более современной виртуальной машиной *ART* (Android Run-Time, которая использует технологию AOT, Ahead-Of-Time). В устройствах с ART приложения компилируются в момент установки на устройство, что позволяет снизить

накладные расходы на систему и увеличить общую производительность. Благодаря использованию виртуальной машины для выполнения кода программы, разработчики получают в свое распоряжение уровень абстракции, позволяющий не беспокоиться об особенностях конструкции различных устройств.

- Встроенный web-browser, основанный на open-source движке WebKit, предоставляет возможность использования рендеринга html-ресурсов в разрабатываемых приложениях.
- Оптимизированная графическая подсистема на базе собственной библиотеки для работы с двухмерной графикой, 3D-графика реализована на базе библиотеки OpenGL ES (с возможной поддержкой аппаратного ускорения).
- Реляционная отказоустойчивая встраиваемая СУБД SQLite для хранения и обработки структурированных данных.
- Встроенная поддержка различных аудио- и видеоформатов, а также статических изображений (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF).
- В зависимости от оборудования, может поддерживаться GSM и CDMA телефония, а также технологии беспроводной передачи данных (Bluetooth, NFC, EDGE, 3G и Wi-Fi).
- Поддержка одной или нескольких камер с возможностью фото/видеосъемки, GPS, компаса и акселерометра и прочих сенсоров.
- Мощная среда разработки, включающая, в том числе, эмулятор (виртуальных) устройств с различными (настраиваемыми) характеристиками, инструменты для отладки и профилирования приложений, а также активно развивающуюся *IDE Android Studio*.

Основные компоненты Android



Упрощенно программный стек *Android* можно представить как комбинацию модифицированного под мобильное применение ядра *Linux* и набора библиотек C/C++, которые доступны в фреймворке приложения. Фреймворк обеспечивает управление и функционирование рабочей среды (runtime environment) и приложений.

Описание основных компонентов платформы Android

Приложения (Applications): *Android* поставляется с набором основных приложений, включающих в себя клиент электронной почты, календарь, карты, браузер, программу для управления контактами и другие. Все эти приложения написаны с использованием программирования Java.
Фреймворк: Предоставляя открытую платформу для разработки, *Android* дает разработчикам возможность создавать мощные и инновационные приложения. Разработчики могут использовать самые современные аппаратные средства, получать информацию о текущем местоположении, запускать фоновые службы, устанавливать сигнализацию, добавлять уведомления в строке состояния, и многое, многое другое. Разработчики имеют полный доступ к тем же API, которые используются при создании встроенных приложений. Архитектура

Android спроектирована для упрощения повторного использования программных компонентов, любое приложение может «опубликовать» свои функциональные возможности, а любые другие приложения могут затем использовать эти возможности (с учетом ограничений безопасности, налагаемых фреймворком). Этот же механизм позволяет пользователям при необходимости заменять программные компоненты. Фреймворк включает в себя набор сервисов и систем, лежащих в основе всех приложений:

- Богатый и расширяемый набор *представлений (View)*, которые могут быть использованы для создания приложений, включая списки (ListView), текстовые поля (TextView и EditText), кнопки (Button) и даже встраиваемые веб-браузер (WebView) и карты (MapView); разработчику также доступны возможности модификации, изменения существующих и создания собстенных элементов интерфейса пользователя.
- *Контент-провайдеры (Content Providers)*, которые позволяют приложениям получать доступ к данным из других приложений (например, контактов), либо делиться своими данными .
- *Менеджер ресурсов (Resource Manager)*, обеспечивающий доступ к ресурсам, не являющимся программным кодом, таким, как строки (strings, в т. ч. локализованные), графические ресурсы (drawable) и файлы разметки (Layouts).
- *Менеджер уведомлений (Notification Manager)*, который позволяет всем приложениям отображать пользовательские уведомления в строке состояния мобильного устройства.
- *Менеджер «Активностей» (Activity Manager)*, который управляет жизненным циклом приложений и предоставляет возможности переключения между различными «Активностями».

Библиотеки: *Android* включает в себя набор библиотек C/C++, используемых различными компонентами системы. Фреймворк предоставляет разработчикам возможности всех этих библиотек. Ниже описаны некоторые из основных библиотек:

- *Системная библиотека C (libc)* – основанная на коде libc, заимствованном из BSD, реализация стандартной системной библиотеки, оптимизированная для мобильных устройств на базе ядра *Linux*.
- *Медиа-библиотека* – основанная на базе фреймворка OpenCore корпорации PacketVideo, библиотека обеспечивает работу со многими популярными форматами аудио, видео и изображений.
- *SurfaceManager* – управляет доступом к подсистеме отображения и обеспечивает «бесшовное» наложение 2D- и 3D-графики из нескольких приложений .
- *LibWebCore* – современный движок, используемый как встроенным в *Android* веб-браузером, так и компонентами внутри приложений (WebView).

- *SGL* – основной механизм для отображения двумерной графики .
- *3D-библиотеки* – реализуют API, основанный на OpenGL ES API; эти библиотеки используют либо аппаратное ускорение 3D-графики (при наличии аппаратного акселератора) или встроенный оптимизированный программный растеризатор трехмерной графики.
- *FreeType* – обеспечивает растровый и векторный рендеринг шрифтов
- *SQLite* – мощный и легкий механизм реляционной СУБД, доступный для всех приложений.

Рабочая среда (runtime environment, RTE):

- *Android* включает в себя набор библиотек, которые обеспечивает большинство runtime-функций, доступных в основных библиотеках языка программирования Java.
- Каждое приложение *Android* выполняется в собственном процессе, со своим собственным экземпляром виртуальной машины *Dalvik (*или *ART)*. ВМ *Dalvik* была спроектирована и написана таким образом, чтобы внутри мобильного устройства могли эффективно работать несколько виртуальных машин. Виртуальная машина *Dalvik* может выполнять программы в исполняемом формате **DEX** (**D**alvik **x**ecutable). Данный формат оптимизирован для использования минимального объема памяти. Исполняемый файл с расширением **.dex** создается путем компиляции классов Java с помощью инструмента dx, входящего в состав *Android SDK*. При использовании *IDE Android Studio* компиляция классов Java в формат .dex происходит автоматически при сборке приложения.
- Виртуальная машина полагается на исполнении ядром *Linux* основных системных функций, таких как многопоточность, ввод/вывод и низкоуровневое управления памятью.

Ядро *Linux*: *Android* использует ядро *Linux* версии 3+ для предоставления основных системных сервисов, таких как обеспечение безопасности, управление памятью, управление процессами, сетевой стек и работа с драйверами. Ядро также выступает как дополнительный уровень абстракции между аппаратным обеспечением мобильного устройства и остальной частью программного стека.

Безопасность и полномочия (Permissions)

Как было сказано выше, инструмент dx из *Android SDK* компилирует приложения, написанные на Java, в исполняемый формат виртуальной машины Dalvik. Помимо непосредственно исполняемых файлов, в состав приложения *Android* входят прочие вспомогательные компоненты (такие, например, как файлы с данными и файлы ресурсов). *SDK* упаковывает все необходимое для установки приложения в файл с расширением **.apk** (a*ndroid* package). Весь код в одном файле .apk считается *одним приложением* и этот файл используется для установки данного приложения на устройствах с ОС *Android*.

После установки на устройства, каждое приложение для *Android* живет в своей собственной изолированной «песочнице» (среде безопасности):

- Система *Android* – это многопользовательская *Linux*-система, в которой каждое приложение работает с правами уникального пользователя. В *Android* реализована характерная для Unix-подобных ОС система базовая безопасности и управления доступом, в которой
  - а) все в системе является файлом, который обязательно принадлежит какому-то пользователю (имеет соответствующий User ID, UID) и
  - б)любой процесс в системе обязательно работает с правами какого-то пользователя (тоже имеет свой UID); сопоставляя UID'ы процесса и файла, система безопасности Unix разрешает или запрещает запрашиваемую процессом операцию с конкретным файлом.
- По умолчанию система присваивает каждому приложению уникальный UID (этот идентификатор используется только операционной системой и неизвестен приложению). Система устанавливает права доступа для всех файлов приложения таким образом, что только процессы с UID'ом, установленным для данного приложения, могут получить к ним доступ.
- Каждый процесс работает внутри своей собственной виртуальной машины, так что код приложения исполняется изолированно от других приложений.
- По умолчанию, каждое приложение выполняется в собственном процессе *Linux*. *Android* создает процесс, когда требуется выполнить какой-либо из компонентов приложения, и уничтожает процесс, когда он больше не нужен или когда система должна предоставить память другим приложениям.

С помощью описанного механизма *Android* реализует принцип наименьших привилегий: каждое приложение по умолчанию имеет доступ только к тем компонентам, которые требуется для выполнения его задач, и не более. Это создает очень безопасную среду, в которой приложение не может получить доступ к частям системы, для которых оно не имеет разрешения.
Тем не менее, для приложений существуют механизмы для обмена данными с другими приложениями и для доступа к системным сервисам:
Двум (или более, при необходимости) приложениям можно присвоить один и тот же UID, в этом случае они смогут получать доступ к файлам друг друга. Чтобы сэкономить системные ресурсы, приложения с одним и тем же UID можно также настроить на исполнение в одном и том же процессе *Linux* в одной и той же виртуальной машине (при этом приложения должны быть подписаны и одним и тем же сертификатом разработчика).
Приложение может запросить разрешение (Permissions) на доступ к данным в устройстве, таким как контакты пользователя, SMS сообщения, сменные

носители информации (SD карты), камера, Bluetooth, и многое другое. Все подобные заявки на доступ к ресурсам проверяются и разрешаются пользователем в ОС *Android* во время установки приложения. Если приложение установлено, пользователя больше не будут спрашивать о пересмотре этих полномочий.
Необходимые приложению полномочия (как и многое другое) указываются в файле Манифеста приложения (*AndroidManifest.xml*). Этот файл обязательно должен присутствовать для любого приложения *Android*. *Android SDK* предоставляет разработчику богатый арсенал средств для управления содержимым этого файла. Кроме того, поскольку файл манифеста является обычным XML-файлом, необходимую информацию в нем можно указывать с помощью обычного редактора (*SDK* и в этом случае позволяет редактировать этот файл с большим комфортом).
Чтобы использовать ресурсы системы и других приложений, требуется добавить тэги *<uses-permission>* в манифест своего приложения, указывая требуемые полномочия. Стандартные имена для описания полномочий, используемых системными Активностями(Activity) и сервисами в *Android* можно найти в классе *Android.Manifest.permission* на сайте: http://developer.android.com/reference/android/Manifest.permission.html
Ниже показан пример объявления запрашиваемого приложением полномочия для доступа к состоянию сетевых соединений:

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

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

    <application
       android:allowBackup="true"
       android:icon="@mipmap/ic_launcher"
       android:label="@string/app_name"
       android:theme="@style/AppTheme">
        . . . . . .
    </application>

</manifest>
```

Компоненты приложения в Android

Типы компонентов

Любое приложение в *Android* строится на основе четырех типов компонентов:

Активность (Activity)

Компонент приложения, предоставляющий интерфейс пользователя (далее — *UI*) в виде экрана, с которым пользователь может взаимодействовать для выполнения каких-либо действий. У каждой Активности имеется собственное окно, в котором она отображает требуемые элементы UI. Обычно это окно полностью занимает экран, но может быть и меньше экрана и располагаться поверх другого окна. Активность, в свою очередь, может использовать для построения UI более мелкие компоненты – *Фрагменты* (*Fragments*). Фрагменты обычно применяются при создании приложений, работающих на большом спектре устройств с самыми разными экранами (как по размеру, так и по разрешению). Фрагменты практически незаменимы при создании приложений со сложными интерфейсами.

Сервис (Service)

Сервисы, в отличие от *Активностей*, не предполагают взаимодействия с пользователем посредством *UI* и предназначены для выполнения фоновых операций. Они являются аналогами *Служб* (*Services*) в Windows и *Демонов* (*Daemons*) в Unix, но, как и любой другой компонент приложения в *Android*, обладают обычно гораздо более коротким жизненным циклом (в хорошо написанном приложении *Сервис* работает ровно столько времени, сколько требуется для выполнения задачи, после чего завершает свою работу).

Широковещательный приемник (Broadcast Receiver)

Широковещательные приемники предназначены для приема широковещательных сообщений, с помощью которых *Android* и приложения извещают всех заинтересованных о наступлении каких-либо событий (изменился уровень заряда батареи, пользователь выключил экран, пришло SMS-сообщение и т. д.).

Контент-провайдер (Content Provider)

Контент-провайдеры предназначены предоставления данных (как правило, структурированных), которыми владеет какое-либо приложение или система, другим приложениям (будучи наследником Unix, *Android* запрещает приложениям доступ к «чужим» данным напрямую). Изоляция потребителя информации от источника данных с помощью Контент-провайдеров также позволяет разработчику инкапсулировать данные внутри приложения-хозяина и предоставлять доступ к ним на базе стандартного интерфейса.

Архитектура приложения

Любой пакет (приложение) для платформы *Android* содержит в себе три части: *код*, *ресурсы* и *мета-информацию* (*Манифест*), необходимую для системы. Код приложения являет собой «активную» часть приложения и строится на основе

описанных выше четырех типов компонентов. Ресурсы, в свою очередь, отделены от кода и могут быть самых разнообразных типов: от иконок и картинок до медиа-файлов и упакованного содержимого БД.
Отделение кода от ресурсов имеет несколько положительных сторон:

- упрощается командная работа над проектом;
- изменение внешнего вида приложения зачастую вообще не требует модификации кода;
- локализация приложения для разных стран требует, как правило, всего лишь перевода строк из UI (хранящихся в строковых ресурсах) на язык нужной страны;
- упрощается адаптация внешнего вида приложения к разнообразным экранам.

Активности (Activity) в Android

При создании экранов графического интерфейса пользователя наследуется класс Activity и используются представления (View) для взаимодействия с пользователем.
Каждая Активность – это экран (по аналогии с Формой), который приложение может показывать пользователям. Чем сложнее создаваемое приложение, тем больше экранов (Активностей) потребуется. При создании приложения потребуется, как минимум, начальный (главный) экран, который обеспечивает основу пользовательского интерфейса приложения. При необходимости этот интерфейс дополняется второстепенными Активностями, предназначенными для ввода информации, ее вывода и предоставления дополнительных возможностей. Запуск (или возврат из) новой Активности приводит к «перемещению» между экранами UI. Большинство Активностей проектируются таким образом, чтобы использовать все экранное пространство, но можно также создавать полупрозрачные или плавающие диалоговые окна.

Создание Активности

Для создания новой Активности наследуется класс Activity. Внутри реализации класса необходимо определить пользовательский интерфейс и реализовать требуемый функционал. Базовый каркас для новой Активности показан ниже:

```
package com.example.myapplication;

import android.app.Activity;
import android.os.Bundle;

public class MyActivity extends Activity {
     /** Вызывается при создании Активности */
     @Override
     public void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
     }
}
```

Базовый класс Activity представляет собой пустой экран, который не особенно полезен, поэтому первое, что вам нужно сделать, это создать пользовательский интерфейс с помощью Представлений (View) и разметки(Layout).
Представления (View) – это элементы UI, которые отображают информацию и обеспечивают взаимодействие с пользователем. Android предоставляет несколько классов разметки (Layout), называемых также *View Groups*,

которые могут содержать внутри себя несколько Представлений, для создания пользовательского интерфейса приложения.
Чтобы назначить пользовательский интерфейс для Активности, внутри обработчика onCreate используется метод setContentView:

```
@Override
public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     TextView textView = new TextView(this);
     setContentView(textView);
}
```

В этом примере в качестве UI для Активности выступает объект класса *TextView*.
При создании реальных приложений чаще применяется метод проектирования, использующий ресурсы приложения, отделенные от кода. Такой подход позволяет создавать приложения, обеспечивающие высокое качество реализации UI, не зависящее от условий работы программы: приложения предлагают удобный для пользователя языковый интерфейс (зависит от локализации), слабо зависят от разрешения и размеров экрана и т. д.). Самое главное, что такая адаптация приложения к новым условиям не требует каких-либо изменений в коде приложения, нужно только обеспечить необходимые ресурсы (картинки, локализованные строки и т. п..). Это стандартный для Android подход показан ниже:

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

Для использования Активности в приложении ее необходимо зарегистрировать в Манифесте путем добавления элемента <activity> внутри узла <application>, в противном случае ее невозможно будет использовать.
Ниже показано, как создать элемент <activity> для Активности MyActivity:

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

В теге <activity> можно добавлять элементы <intent-filter> для указания Намерений (Intent), которые Активность будет отслеживать. Каждый «Фильтр Намерений» определяет одно или несколько действий (action) и категорий (category), которые поддерживаются Активностью. Важно знать, что Активность будет доступна из главного меню запуска приложений только в случае, если в Манифесте для нее указан <intent-filter> для действия MAIN и

категории LAUNCHER, как показано на примере:

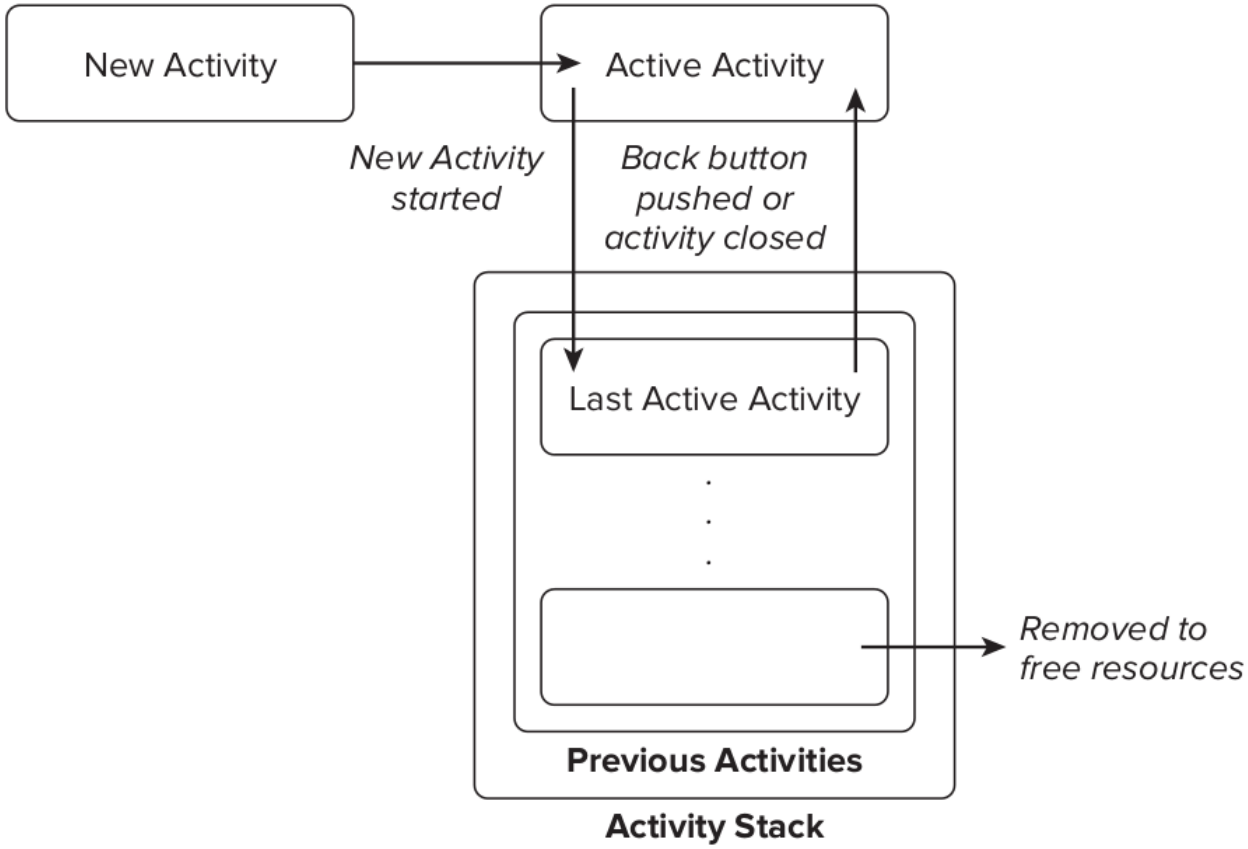
```
<activity android:label="@string/app_name"
     android:name=".MyActivity">
     <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name=
          "android.intent.category.LAUNCHER" />
     </intent-filter>
</activity>
```

Жизненный цикл Активности

Для создания приложений, правильно управляющих ресурсами предоставляющих пользователю удобный интерфейс, важно хорошее понимание жизненного цикла Активности. Это связано с тем, что приложения Android не могут контролировать свой жизненный цикл, ОС сама управляет всеми процессами и, как следствие, Активностями внутри них. При этом, состояние Активности помогает ОС определить приоритет родительского для этой Активности Приложения (Application). А приоритет Приложения влияет на то, с какой вероятности его работа (и работа дочерних Активностей) будет прервана системой.

Стеки Активностей

Состояние каждой Активности определяется ее позицией в стеке (LIFO) Активностей, запущенных в данный момент. При запуске новой Активности представляемый ею экран помещается на вершину стека. Если пользователь нажимает кнопку «назад» или эта Активности закрывается каким-то другим образом, на вершину стека перемещается (и становится *активной*) нижележащая Активность. Данный процесс показан на диаграмме:



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

Состояния Активностей

Активности могут находиться в одном из четырех возможных состояний:

- **Активное (Active)**. Активность находится на переднем плане (на вершине стека) и имеет возможность взаимодействовать с пользователем. Android будет пытаться сохранить ее работоспособность любой ценой, при необходимости прерывая работу других Активностей, находящихся на более низких позициях в стеке для предоставления необходимых ресурсов. При выходе на передний план другой Активности работа данной Активности будет *приостановлена* или *остановлена*.
- **Приостановленное (Paused)**. Активность может быть видна на экране, но не может взаимодействовать с пользователем: в этот момент она приостановлена. Это случается, когда на переднем плане находятся полупрозрачные или плавающие (например, диалоговые) окна. Работа приостановленной Активности может быть прекращена, если ОС необходимо выделить ресурсы Активности переднего плана. Если Активность полностью исчезает с экрана, она *останавливается*.

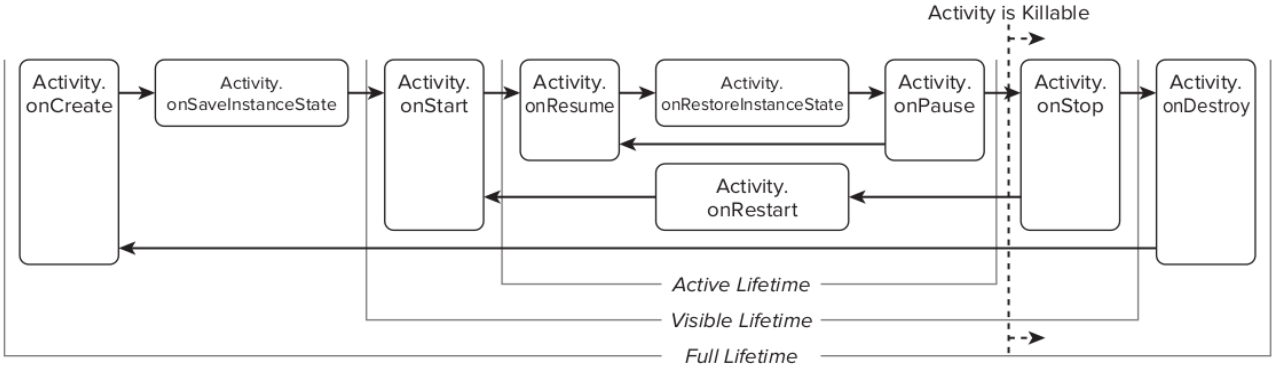
- **Остановленное (Stopped)**. Активность невидима, она находится в памяти, сохраняя информацию о своем состоянии. Такая Активность становится кандидатом на преждевременное закрытие, если системе потребуется память для чего-то другого. При остановке Активности разработчику важно сохранить данные и текущее состояние пользовательского интерфейса (состояние полей ввода, позицию курсора и т. д.). Если Активность завершает свою работу или закрывается, он становится *неактивным*.
- **Неактивное (Inactive)**. Когда работа Активности завершена, и перед тем, как она будет запущена, данная Активности находится в неактивном состоянии. Такие Активности удаляются из стека и должны быть (пере)запущены, чтобы их можно было использовать.

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

Для обеспечения полноценного интерфейса приложения, изменения его состояния должны быть незаметными для пользователя. Меняя свое состояние с приостановленного на остановленное или с неактивного на активное, Активность не должна внешне меняться. При остановке или приостановке работы Активности разработчик должен обеспечить сохранение состояния Активности, чтобы его можно было восстановить при выходе Активности на передний план. Для это в классе Activity имеются обработчики событий, переопределение которых позволяет разработчику отслеживать изменение состояний Активности.

Отслеживание изменений состояния Активности

Обработчики событий класса Activity позволяют отслеживать изменения состояний соответствующего объекта Activity во время всего жизненного цикла:



Ниже показан пример с заглушками для таких методов – обработчиков событий:

```
package com.example.myapplication;

import android.app.Activity;
import android.os.Bundle;

public class MyActivity extends Activity {

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

     // Вызывается после завершения метода onCreate
     // Используется для восстановления состояния UI
     @Override
     public void onRestoreInstanceState(Bundle savedInstanceState)
     {
          super.onRestoreInstanceState(savedInstanceState);
          // Восстановить состояние UI из объекта
          // savedInstanceState.
          // Данный объект также был передан методу onCreate.
     }

     // Вызывается перед тем, как Активность снова
     // становится видимой
     @Override
     public void onRestart(){
          super.onRestart();
          // Восстановить состояние UI с учетом того,
          // что данная Активность уже была видимой.
     }

     // Вызывается когда Активность стала видимой
     @Override
     public void onStart(){
          super.onStart();
          // Проделать необходимые действия для
          // Активности, видимой на экране
     }

     // Должен вызываться в начале видимого состояния.
     // На самом деле Android вызывает данный обработчик только
```

```
    // для Активностей, восстановленных из неактивного состояния
    @Override
    public void onResume(){
        super.onResume();
        // Восстановить приостановленные обновления UI,
        // потоки и процессы, «замороженные, когда
        // Активность была в неактивном состоянии
    }

    // Вызывается перед выходом из активного состояния,
    // позволяя сохранить состояние в объекте savedInstanceState
    @Override
    public void onSaveInstanceState(Bundle savedInstanceState) {
        super.onSaveInstanceState(savedInstanceState);
        // Объект savedInstanceState будет в последующем
        // передан методам onCreate и onRestoreInstanceState
    }


    // Вызывается перед выходом из активного состояния
    @Override
    public void onPause(){
        super.onPause();
        // «Заморозить» обновления UI, потоки или
        // «трудоемкие» процессы, ненужные, когда Активность
        // не на переднем плане
    }

    // Вызывается перед выходом из видимого состояния
    @Override
    public void onStop(){
        super.onStop();
        // «Заморозить» обновления UI, потоки или
        // «трудоемкие» процессы, ненужные, когда Активность
        // не на переднем плане.
        // Сохранить все данные и изменения в UI, так как
        // процесс может быть в любой момент убит системой
        // без объявления войны
    }

    // Вызывается перед уничтожением активности
    @Override
    public void onDestroy(){
        super.onDestroy();
        // Освободить все ресурсы, включая работающие потоки,
        // соединения с БД и т. д.
    }
}
```

Лабораторная работа «Отслеживание состояний Активности»

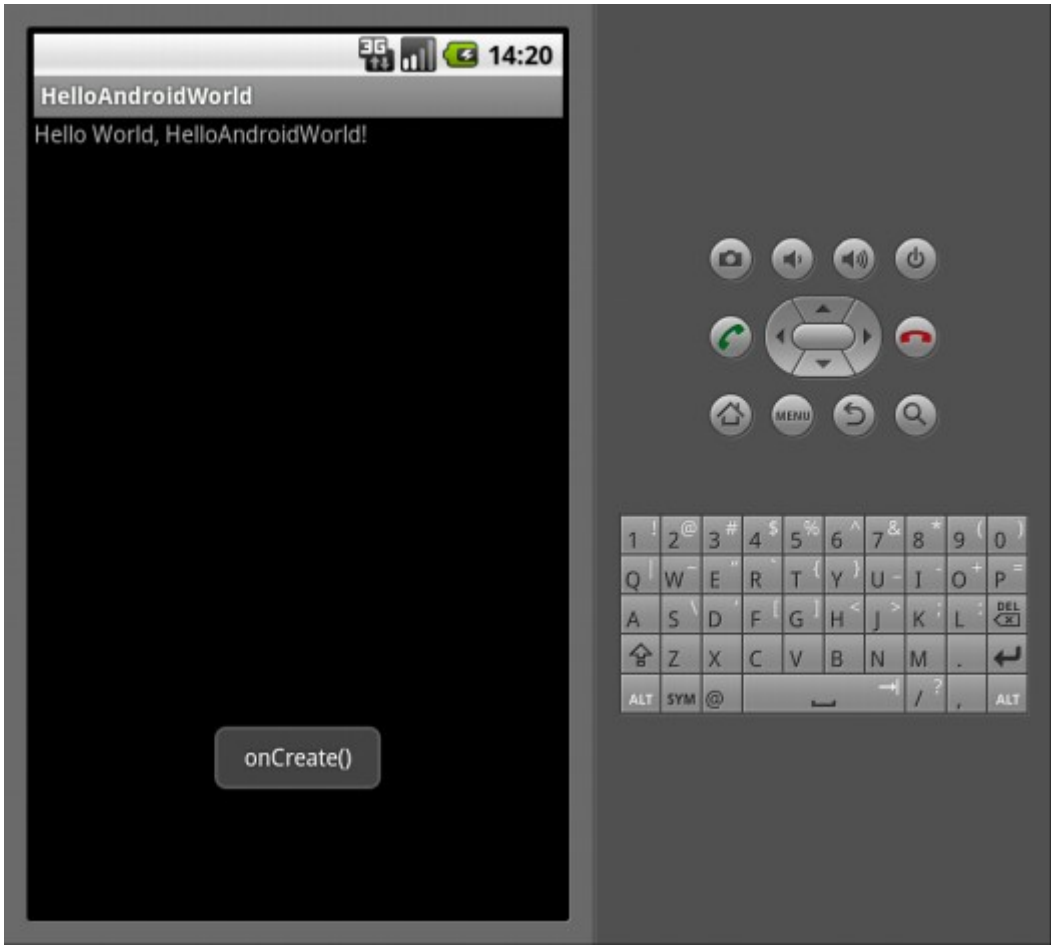
1. В имеющемся проекте *HelloAndroidWorld* откройте в редакторе класс *HelloAndroidWorld.java.*
2. Переопределите методы *onPause*, *onStart*, *onRestart* для класса и внесите изменения в метод *onCreate*:

```
package com.example.helloandroidworld;

import android.app.Activity;
import android.os.Bundle;
import android.widget.Toast;

public class HelloAndroidWorld extends Activity {

     @Override
     public void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.main);
          Toast.makeText(this, "onCreate()", Toast
               .LENGTH_LONG).show();
     }

     @Override
     protected void onPause() {
          super.onPause();
          Toast.makeText(this, "onPause()", Toast
               .LENGTH_LONG).show();
     }

     @Override
     protected void onRestart() {
          super.onRestart();
          Toast.makeText(this, "onRestart()", Toast
               .LENGTH_LONG).show();
     }


     @Override
     protected void onStart() {
          super.onStart();
          Toast.makeText(this, "onStart()", Toast
               .LENGTH_LONG).show();
     }
}
```

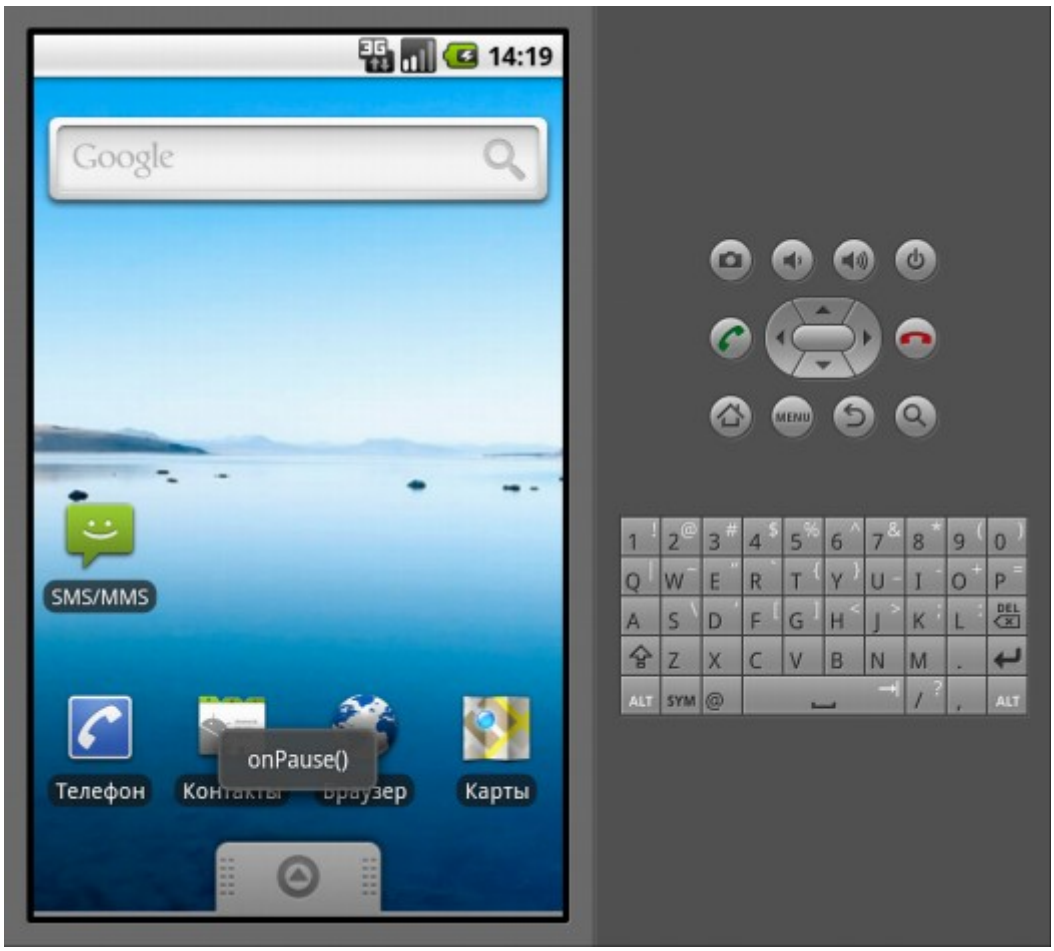
Обратите внимание, что в методах, *восстанавливающих* состояние, вызов метода родительского класса производится *до* ваших действий, а в

методах, состояние *сохраняющих – после* ваших действий.

3. Запустите приложение на исполнение в эмуляторе (Ctrl+F11, Android Project) и убедитесь, что при изменении состояния приложения HelloAndroidWorld на экране эмулятора появляются соответствующие уведомления типа Toast, как показано на снимках:



4. Поэкспериментируйте в приложением, чтобы выяснить, при каких условиях вызываются реализованные обработчики событий.


14:19
Google
SMS/MMS
onPause()
Телефон
Карты
MENU

Ресурсы

Отделение ресурсов от кода программы

Независимо от используемой среды разработки, весьма разумно отделять используемые приложением ресурсы от кода. Внешние ресурсы легче поддерживать, обновлять и контролировать. Такая практика также позволяет описывать альтернативные ресурсы для поддержки вашим приложением различного оборудования и реализовывать локализацию приложения. Приложения Android используют разнообразные ресурсы из внешних (по отношению к коду) файлов, от простых (строки и цвета) до более сложных (изображения, анимации, визуальные стили). Чрезвычайно полезно также отделять от кода такие важные ресурсы, как разметки экранов (Layout), используемые в Активностях. Android автоматически выбирает наиболее подходящие варианты из дерева ресурсов приложения, содержащие разные значения для разных аппаратных конфигураций, языков и регионов, не требуя при этом ни единой строчки кода.

Создание ресурсов

Ресурсы приложения хранятся в каталоге **res** в дереве каталогов проекта. Плагин ADT автоматически создает каталог res с подкаталогами **values**, **layout** и **drawable-***, в которых хранятся, соответственно: строковые константы, разметка по умолчанию и иконка приложения для разных плотностей пикселей на экране.
Для девяти главных типов ресурсов используются разные подкаталоги каталога **res**, это:

- простые значения
- изображения
- разметка
- анимация
- стили
- меню
- настройки поиска
- XML
- «сырые» данные

При сборке пакета .apk эти ресурсы максимально эффективно компилируются и включаются в пакет.
Для работы с ресурсами внутри кода плагин ADT автоматически генерирует файл класса **R**, содержащий ссылки на все ресурсы. Имена файлов ресурсов могут содержать только латинские буквы в нижнем регистре, подчеркивания и точки.

Простые значения (values)

Android поддерживает следующие типы значений: строки, цвета, размеры и массивы (строковые и целочисленные). Эти данные хранятся в виде XML-файла в каталоге **res/values**. Ниже показан пример подобного файла:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
     <string name="app_name">To Do List</string>
     <color name="app_background">#FF0000FF</color>
     <dimen name="default_border">5px</dimen>
     <array name="string_array">
          <item>Item 1</item>
          <item>Item 2</item>
          <item>Item 3</item>
     </array>
     <array name="integer_array">
          <item>3</item>
          <item>2</item>
          <item>1</item>
     </array>
</resources>
```

В примере содержатся все доступные типы простых значений, но на самом деле каждый тип ресурсов хранится в отдельном файле, например, **res/values/arrays.xml** содержит массивы, а **res/values/strings.xml** – строковые константы.

Строки

Строковые константы определяются с помощью тэга <*string*>, как показано на примере:

```
<string name="greeting_msg">Превед!</string>
```

Для выделения текста в строках можно использовать HTML-тэги **<b>**, **<i>** и **<u>**.
Пример:

```
<string name="greeting_msg"><b>Превед</b>, дарлинг!</string>
```

При необходимости использования данной строки в коде программы используется вышеупомянутый класс **R**:

```
String greeting = getString(R.string.greeting_msg);
```

Цвета

Для описания цветов используется тэг <*color*>. Значения цвета указываются в шестнадцатеричном виде в одном из следующих форматов:

- #RGB
- #ARGB
- #RRGGBB
- #AARRGGBB

В примере показаны описания полупрозрачного красного цвета и непрозрачного зеленого:

```
<color name="transparent_red">#77FF0000</color>
<color name="opaque_green">#0F0</color>
```

Лабораторная работа «Использование значений строк и цветов»

1. В имеющемся проекте *HelloAndroidWorld* создайте файл *colors* в каталоге *res/values*.
2. Отредактируйте содержимое файла **res/values/colors.xml**:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="view_bkg_color">#FF0</color>
    <color name="screen_bkg_color">#F88</color>
    <color name="text_color">#8004</color>
</resources>
```

3. Отредактируйте содержимое файла **res/values/strings.xml**:

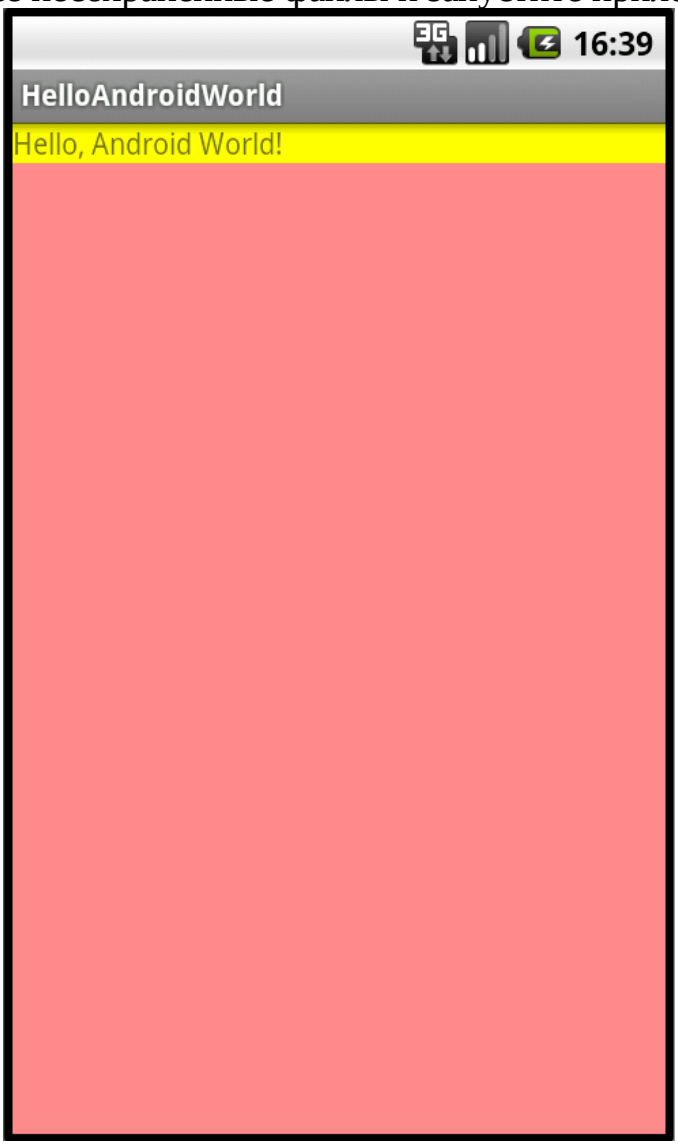
```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="hello">Hello, Android World!</string>
    <string name="app_name">HelloAndroidWorld</string>

</resources>
```

4. Внесите изменения в файл разметки **res/layout/main.xml**:

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

```
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"
        android:background="@color/view_bkg_color"
        android:textColor="@color/text_color"/>
</LinearLayout>
```

5. Сохраните все несохраненные файлы и запустите приложение:



6. Убедитесь, что внешний вид приложения ~~стал невыносимо мерзким~~ изменился в соответствии с определенными нами ресурсами.
7. Поэкспериментируйте со значениями цветов, прозрачностью и HTML-тэгами в строковых константах для изменения внешнего вида приложения.

Размеры

Ссылки на размеры чаще всего встречаются внутри ресурсов со стилями и

разметкой, например, при указании толщины рамки или величины шрифта. Для описания размеров используется тэг *<dimen>* с указанием вида размерности:

- **px** – реальные экранные пикселы
- **in** – физические дюймы
- **pt** – 1/72 дюйма, вычисляется из физического размера экрана
- **mm –** физические миллиметры, вычисляется из физического размера экрана
- **dp** – «независимые» от плотности экрана пиксели, равны одному пикселю при эталонной плотности 160 dpi; можно также указывать как **dip**; чаще всего используются для указания размеров рамок и полей
- **sp** – «независимые» от масштаба пиксели, аналогичны **dp**, но учитывают также пользовательские настройки размеров шрифта (крупный, мелкий, средний), поэтому рекомендуются для описания шрифтов

Пример описания «большого» шрифта и «стандартной» рамки:

```
<dimen name="standard_border">5dp</dimen>
<dimen name="large_font_size">16sp</dimen>
```

Визуальные стили и темы

Стили и темы позволяют поддерживать единство внешнего вида приложения с помощью атрибутов, используемых Представлениями (View), чаще всего это цвета и шрифты. Внешний вид приложения легко меняется при изменении стилей (тем оформления) в Манифесте приложения.
Для создания стиля используется тэг *<style>* с атрибутом *name*, содержащий элементы *<item>*, каждый из которых, в свою очередь, также имеет атрибут *name*, указывающий тип параметра (например, цвет или размер). Внутри элемента *<item>* хранится значение параметра:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
     <style name="StyleName">
          <item name="attributeName">attributeValue</item>
          [ … еще элементы <item> … ]
     </style>
</resources>
```

В тэге <style> можно указать атрибут parent, что делает возможным «наследование» стилей при необходимости внести в новый стиль незначительные отличия от имеющегося:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
     <style name="NormalText">
          <item name="android:textSize">14sp</item>
          <item name="android:textColor">#111</item>
     </style>
     <style name="SmallText" parent="NormalText">
          <item name="android:textSize">8sp</item>
     </style>
</resources>
```

Изображения

Ресурсы *Drawable* содержат растровые изображения. Это могут быть сложные составные ресурсы, такие, как *LevelListDrawables* и *StateListDrawables*, описываемые в формате XML, а также растягиваемые растровые изображения *NinePatch*. Ресурсы Drawable хранятся в каталогах **res/drawable-*** виде отдельных файлов. Идентификаторами для таких ресурсов служат имена в нижнем регистре (без расширения). Поддерживаются форматы PNG (предпочтительный), JPEG и GIF.

Разметка

Благодаря использованию ресурсов с разметкой (layout) разработчик имеет возможность отделить логику программы от ее внешнего вида. Разметку, определенную в файле формата XML, можно загрузить для использования в Активности методом *setContentView*, как это реализовано в нашем приложении в методе *onCreate*:

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

Каждый ресурс с разметкой хранится в отдельном файле в каталоге **res/layout**. Имя файла используется как идентификатор данного ресурса (как обычно, без расширения). При создании нашего учебного приложения мастер создания новых проектов создал для нас файл res/layout/main.xml, который мы уже редактировали:

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

    <TextView
        android:layout_width="fill_parent"
```

```
        android:layout_height="wrap_content"
        android:background="@color/view_bkg_color"
        android:text="@string/hello"
        android:textColor="@color/text_color" />

</LinearLayout>
```

Данный файл содержит разметку *LinearLayout*, которая является контейнером для элемента *TextView*, отображающее содержимое строки с именем *hello*, описанную в ресурсе *strings*.

Анимация

Android поддерживает два вида анимации: пошаговую анимацию, последовательно выводящую на экран изображения с заданной длительностью, и анимацию, основанную на расчете промежуточных кадров, в этом случае применяются различные преобразования – вращения, растягивания, перемещения и затемнения. Все эти трансформации описываются в XML-файле в каталоге **res/anim**. Пример файла анимации, в котором целевой элемент одновременно поворачивается на 270 градусов, сжимается и постепенно исчезает:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
     android:interpolator="@android:anim/accelerate_interpolator">
     <rotate
          android:fromDegrees="0"
          android:toDegrees="270"
          android:pivotX="50%"
          android:pivotY="50%"
          android:startOffset="500"
          android:duration="1000" />

     <scale
          android:fromXScale="1.0"
          android:toXScale="0.0"
          android:fromYScale="1.0"
          android:toYScale="0.0"
          android:pivotX="50%"
          android:pivotY="50%"
          android:startOffset="500"
          android:duration="500" />

     <alpha
          android:fromAlpha="1.0"
          android:toAlpha="0.0"
          android:startOffset="500"
          android:duration="500" />
```

```
</set>
```

Ресурс , описывающий пошаговую анимацию, хранится в каталоге **res/drawable**. В следующем примере описана анимация, основанная на последовательном отображении шести изображений поезда, каждое из которых (кроме последнего) отображается в течении 200 миллисекунд. Разумеется, для использования такой анимации нужны ресурсы с изображениями (Drawable), находящимися в этом же каталоге с именами (как вариант, в формате PNG) *train1.png .. train6.png*.

```
<animation-list
     xmlns:android="http://schemas.android.com/apk/res/android"
     android:oneshot="false">
          <item android:drawable="@drawable/train1"
               android:duration="200" />
          <item android:drawable="@drawable/train2"
               android:duration="200" />
          <item android:drawable="@drawable/train3"
               android:duration="200" />
          <item android:drawable="@drawable/train4"
               android:duration="200" />
          <item android:drawable="@drawable/train5"
               android:duration="200" />
          <item android:drawable="@drawable/train6"
               android:duration="1500" />
</animation-list>
```

Подробную информацию о возможностях анимации в Android можно найти здесь:
http://developer.android.com/guide/topics/graphics/animation.html

Меню

Ресурсы меню могут использоваться для описания как главного меню Активности, так и контекстного, появляющегося при длительном нажатии на какой-либо элемент пользовательского интерфейса (разумеется, если это поддерживается и имеет смысл для вашего приложения). Меню, описанное в формате XML, загружается в приложение с помощью метода *inflate* системного сервиса MenuInflater. Обычно это происходит внутри метода *onCreateOptionsMenu* (для главного меню) или *onCreateContextMenu* (для контекстного меню), переопределенных в Активности. Каждый экземпляр меню описывается в отдельном XML-файле в каталоге **res/menu**. Как обычно, имена файлов (без расширений) становятся именами ресурсов. Ниже приведен пример простого ресурса с меню, имеющего три пункта: Refresh, Settings и Quit:

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

```
          android:title="Refresh" />
     <item android:id="@+id/menu_settings"
          android:title="Settings" />
     <item android:id="@+id/menu_quit"
          android:title="Quit" />
</menu>
```

Каждый из этих пунктов меню имеет уникальный идентификатор (*menu_refresh*, *menu_settings* и *menu_quit*), позволяющий в дальнейшем обработчику меню определить, какой из пунктов был выбран пользователем.

Использование внешних ресурсов в коде приложения

Доступ к ресурсам в коде осуществляется с помощью автоматически генерируемого класса **R**, точнее, его подклассов. Например, класс **R** в нашем проект выглядит так:

```
package com.example.helloandroidworld;

public final class R {
    public static final class attr {
    }
    public static final class color {
        public static final int screen_bkg_color=0x7f040001;
        public static final int text_color=0x7f040002;
        public static final int view_bkg_color=0x7f040000;
    }
    public static final class mipmap {
        public static final int ic_launcher=0x7f020000;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f050001;
        public static final int hello=0x7f050000;
    }
}
```

Члены классов с именами, соответствующими ресурсам, являются идентификаторами в таблице ресурсов, а не самими экземплярами ресурсов. Некоторые методы и конструкторы могут принимать в качестве параметров идентификаторы ресурсов, в этом случае их можно использовать напрямую:

```
setContentView(R.layout.main);
Toast.makeText(this, R.string.awesome_error,
Toast.LENGTH_LONG).show();
```

В случае, если необходим *экземпляр* ресурса, требуется доступ к таблице ресурсов, осуществляемый с помощью экземпляра класса *Resources*. Этот класс содержит геттеры для всех видов ресурсов, при этом в качестве параметров используются идентификаторы ресурсов из класса **R**:

```
// Получаем доступ к таблице ресурсов
Resources r = getResources();

// и получаем необходимые экземпляры ресурсов
CharSequence greetingMsg = r.getText(R.string.greeting_message);

Drawable icon = r.getDrawable(R.drawable.app_icon);

int opaqueBlue = r.getColor(R.color.opaque_blue);

float borderWidth = r.getDimension(R.dimen.standard_border);

String[] stringArray = r.getStringArray(R.array.string_array);

int[] intArray = r.getIntArray(R.array.integer_array);
```

Использование ресурсов внутри ресурсов

Для обращение к одному ресурсу внутри описания другого ресурса используется следующая нотация:

```
attribute="@[packagename:]resourcetype/resourceidentifier"
```

Имя пакета используется только при обращении к ресурсам из другого пакета, для обращения к своим ресурсам его указывать не требуется. В нашем случае такую нотацию можно увидеть, например, при описании элемента разметки *TextView* в файле **res/layout/main.xml**:

```
<TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="@color/view_bkg_color"
        android:text="@string/hello"
        android:textColor="@color/text_color" />
```

Аналогичным образом осуществляется доступ к системным ресурсам, в качестве имени пакета при этом указывается @android:

```
<EditText
        android:id="@+id/myEditText"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
```

```
        android:text="@android:string/ok"
        android:textColor="@android:color/primary_text_light" />
```

Обратите внимание на атрибут **android:id="@+id/myEditText"**. Такая запись позволяет присвоить идентификатор вашему компоненту ресурса (в данном случае элементу
разметки) и в дальнейшем использовать этот идентификатор для получения экземпляра ресурса.

Ссылки на текущие визуальные стили позволяют использовать действующие в настоящий момент атрибуты стилей, вместо того, чтобы заново их определять. Для указания ссылки на такой ресурс используется символ **?** Вместо **@**:

```
<EditText
        android:id="@+id/myEditText"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/edit_text_hint"
        android:backgroundColor="?android:backgroundColor" />
```

Локализация приложения с помощью внешних ресурсов

Одно из основных преимуществ использования внешних по отношению к коду приложения ресурсов – возможность использования механизма автоматического выбора ресурсов. Пользуясь описанным ниже механизмом, можно создавать индивидуальные ресурсы для различных аппаратных конфигураций, языков, регионов и т. д. Android во время выполнения приложения сам выберет наиболее подходящие ресурсы.

Для индивидуальной настройки приложения доступны следующие возможности:

- **MCC** (Mobile Country Code) и **MNC** (Mobile Network Code)
- **Язык и регион**. Например, *en-rUS* для английского языка в американском регионе (маленькая **r** – от «**r**egion»), *ru* для русского
- **Размер экрана** (*small, medium* или *large*)
- **«Широкоформатность» экрана** (*long* или *notlong*)
- **Ориентация экрана** (*port, land* или *square*)
- **Плотность пикселей на экране** (*ldpi, mdpi, hdpi* или *nodpi*)
- **Тип сенсорного экрана** (*notouch, stylus* или *finger*)
- **Доступность клавиатуры** (*keysexposed, keyshidden* или *keyssoft*)
- **Тип ввода** (*nokeys, qwerty* или *12key*)
- **Способ навигации** (*nonav, dpad, trackball* или *wheel*)

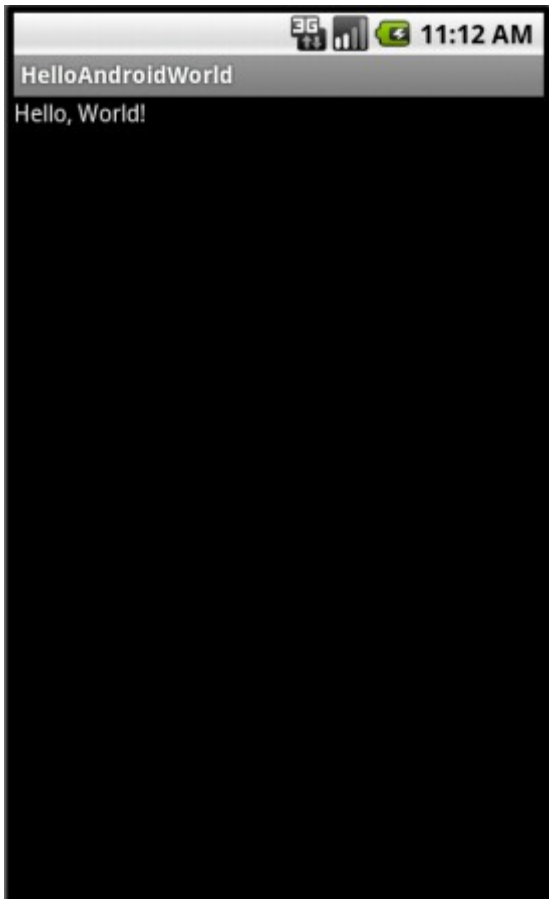
Альтернативные ресурсы располагаются в подкаталогах каталога **res**, при этом используются модификаторы стандартных имен подкаталогов с ресурсами. Например, файл, содержащий строковые константы для русского

языка, будет располагаться по следующему пути: **res/values-ru/strings.xml** Модификатором в данном случае является суффикс **-ru**, добавленный к имени каталога **values**.

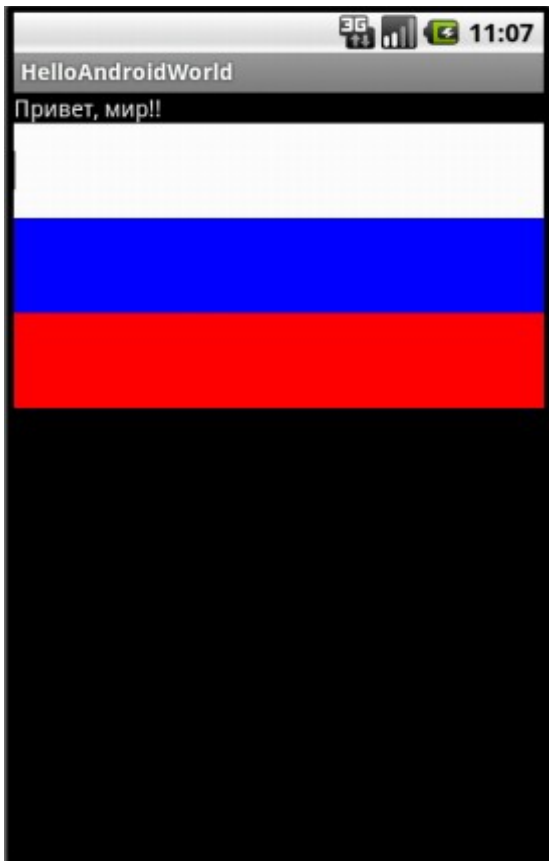
Более подробно об актуальных типах ресурсов, их применении и используемых модификаторах (суффиксах) можно узнать из первоисточника: http://developer.android.com/guide/topics/resources/providing-resources.html

Лабораторная работа «Локализация приложения»

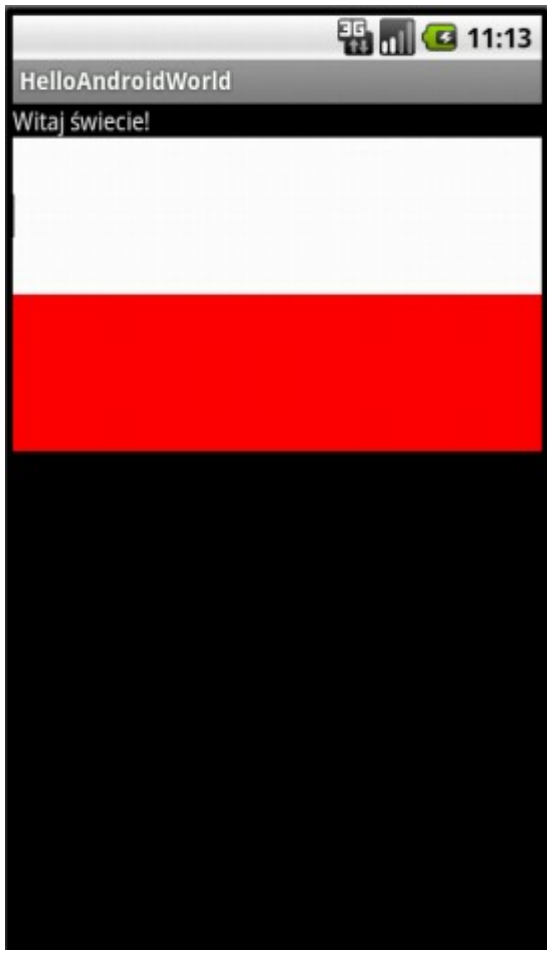
1. Измените ресурсы приложения *HelloAndroidWorld* так, чтобы оно выглядело следующим образом (на виртуальном устройстве должен быть установлен английский язык):



2. Добавьте нужные подкаталоги и ресурсы в каталог **res** проекта *HelloAndroidWorld*, чтобы при настройке русского языка приложение меняло свой вид на следующий:



3. Проделайте те же действия для польского языка:



4. Восстановите языковые настройки на виртуальном устройстве.

Лабораторная работа «Использование анимации»

1. Создайте новый проект **AnimSample**.
2. В каталоге **res** создайте каталог **anim**, а в нем файл с именем **ship_anim.xml**, описывающий анимацию. Отредактируйте файл, чтобы он имел следующее содержимое:

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

    <rotate
        android:duration="3333"
        android:fromDegrees="0"
        android:pivotX="50%"
        android:pivotY="50%"
        android:repeatCount="infinite"
        android:repeatMode="reverse"
        android:toDegrees="1080" />

    <translate
        android:duration="1900"
        android:fromXDelta="-50%p"
        android:repeatCount="infinite"
        android:repeatMode="reverse"
        android:toXDelta="50%p" />
    <translate
        android:duration="1300"
        android:fromYDelta="-50%p"
        android:repeatCount="infinite"
        android:repeatMode="reverse"
        android:startOffset="123"
        android:toYDelta="50%p" />

    <alpha
        android:duration="500"
        android:fromAlpha="1.0"
        android:repeatCount="infinite"
        android:repeatMode="reverse"
        android:toAlpha="0.3" />

    <scale
        android:duration="10000"
        android:fromXScale="0.0"
        android:fromYScale="0.0"
        android:pivotX="50%"
        android:pivotY="50%"
        android:repeatCount="infinite"
```

```
        android:repeatMode="reverse"
        android:toXScale="2.5"
        android:toYScale="2.5" />

</set>
```

3. Добавьте рисунок, к которому будет применяться анимация (файл **lander_plain.png**), в каталог **res/drawable-mdpi**.
4. В файле разметки **res/layout/main.xml** замените элемент **TextView** на **ImageView** со следующими атрибутами:

```
<ImageView
        android:id="@+id/shipView"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:src="@drawable/lander_plain" />
```

5. В конце метода **onCreate** Активности (она в этом проекте одна) добавьте следующие строки:

```
ImageView ship = (ImageView) findViewById(R.id.shipView);
Animation shipAnim = AnimationUtils.loadAnimation(this,
            R.anim.ship_anim);
ship.startAnimation(shipAnim);
```

6. Запустите проект.

Класс Application

Создание собственной реализации класса Application дает возможность:

- контролировать состояние приложения
- передавать объекты между программными компонентами
- поддерживать и контролировать ресурсы, которые используются в нескольких компонентах одного приложения

При создании операционной системой процесса, в котором будет исполняться программа (с точки зрения Unix-подобных систем, процессы являются *контейнерами*, в которых исполняются программы) создается экземпляр класса Application, который описан в Манифесте приложения. Таким образом, Application по своей природе является синглтоном (s*ingleton*) и должен быть правильно реализован, чтобы предоставлять доступ к своим методам и переменным.

Наследование и использование класса Application

Ниже показан каркас для наследования класса Application и использования его в качестве синглтона:

```
import android.app.Application;
import android.content.res.Configuration;

public class MyApplication extends Application {

     private static MyApplication singleton;

     // Возвращает экземпляр данного класса
     public static MyApplication getInstance() {
          return singleton;
     }

     @Override
     public final void onCreate() {
          super.onCreate();
          singleton = this;
     }
}
```

После создания реализация класса Application должна быть зарегистрирована с Манифесте приложения, для этого используется элемент <*application*>:

```
<application
     android:icon="@drawable/icon"
     android:name="MyApplication">
     [... вложенные элементы ...]
</application>
```

Теперь при запуске приложения будет создавать экземпляр вашей реализации класса Application. Если есть необходимость хранения состояния приложения и глобальных ресурсов, реализуйте соответствующие методы в вашей реализации класса и используйте их в компонентах вашего приложения:

```
SomeObject value = MyApplication.getInstance()
     .getGlobalStateValue();
MyApplication.getInstance().setGlobalStateValue(someObjectValue);
```

Такой подход может быть эффективен при передаче объектов между слабосвязанными частями программы и для контроля за общими ресурсами и состоянием приложения.

Обработка событий жизненного цикла приложения

Аналогично обработчикам событий класса Activity, обработчики событий класса Application так же можно переопределять для управления реакцией приложения на те или иные события жизненного цикла:

```
import android.app.Application;
import android.content.res.Configuration;

public class MyApplication extends Application {

     @Override
     public void onConfigurationChanged(Configuration newConfig) {
          super.onConfigurationChanged(newConfig);
     }

     @Override
     public void onCreate() {
          super.onCreate();
     }

     @Override
     public void onLowMemory() {
          super.onLowMemory();
     }

     @Override
     public void onTerminate() {
          super.onTerminate();
     }
}
```

Назначение этих методов следующее:

- **onCreate**: вызывается при создании приложения, переопределяется для создания и инициализации свойств, в которых хранятся состояния приложения или глобальные ресурсы.
- **onTerminate**: может быть вызван при преждевременном завершении работы приложения (но может и не быть вызван: если приложение закрывается ядром, чтобы освободить ресурсы для других программ, onTerminate вызван не будет).
- **onLowMemory**: предоставляет возможность приложениям освободить дополнительную память (по-хорошему :), когда ОС не хватает ресурсов. Этот метод переопределяется для того, чтобы очистить кэш или освободить ненужные в данный момент ресурсы.
- **onConfigurationChanged**: переопределяется, если необходимо отслеживать изменения конфигурации на уровне приложения(такие, например, как поворот экрана, закрытие выдвижной клавиатуры устройства и т. п.).

Понятие контекста

Класс **android.context.Context** представляет из себя интерфейс для доступа к глобальной информации об окружении приложения. Это абстрактный класс, реализация которого обеспечивается системой Android. Context позволяет получить доступ к специфичным для данного приложения ресурсам и классам, а также для вызова операций на уровне приложения, таких, как запуск Активностей, отправка широковещательных сообщений, получение намерений (Intent) и прочее.
Данный класс также является базовым для классов Activity, Application и Service. Получить доступ контексту можно с помощью методов *getApplicationContext*, *getContext*, *getBaseContext*, а также просто с помощью свойства *this* (изнутри Активности или Сервиса). Мы уже воспользовались последним способом при вызове статического метода *makeText* класса *Toast*, который получает контекст в качестве первого параметра:

```
Toast.makeText(this, "onCreate()", Toast.LENGTH_LONG).show();
```

Типичное использование контекста может быть таким:
Создание новых объектов (Представлений, Адаптеров и т. д.):

```
TextView myTextView = new TextView(getContext());
ListAdapter listAdapter = new SimpleCursorAdapter(
     getApplicationContext(), ...);
```

Доступ к стандартным глобальным ресурсам:

```
context.getSystemService(LAYOUT_INFLATER_SERVICE);
prefs = getApplicationContext()
```

```
    .getSharedPreferences("PREFS",MODE_PRIVATE);
```

Неявное использование глобальных компонентов системы:

```
cursor = getApplicationContext()
    .getContentResolver().query(uri, ...);
```

Пользовательский интерфейс

Основные понятия и связи между ними

- **Представления (View)** являются базовым классом для все визуальных элементов UI (элементов управления (*Control*) и виджетов (*Widget*)). Все эти элементы, в том числе и разметка (*Layout*), являются расширениями класса **View**.
- **Группы Представлений (ViewGroup)** – потомки класса *View;* могут содержать в себе несколько дочерних *Представлений*. Расширение класса ViewGroup используется для создания сложных Представлений, состоящих из взаимосвязанных компонентов. Класс *ViewGroup* также является базовым для различных Разметок (Layout).
- **Активности (Activity)** – отображаемые экраны или окна (с точки зрения построения UI), являются «андроидными эквивалентами» форм. Для отображения UI Активности используют Представления (как правило, Разметки (*Layout*)).

Для создания приложений с уникальными интерфейсом разработчику иногда приходится расширять и модифицировать стандартные Представления, комбинируя их со стандартными.

Представления (View)

Android предоставляет разработчику возможность использования богатого набора готовых Представлений:

- **TextView**. Стандартный элемент, предназначенный для вывода текста. Поддерживает многострочное отображение, форматирование и автоматический перенос.
- **EditText**. Редактируемое поле для ввода текста. Поддерживает многострочный ввод, перенос слов на новую строку и и текст подсказки.
- **ListView**. Группа представлений (*ViewGroup*), которая формирует вертикальный список элементов, отображая их в виде строк внутри списка. Простейший объект *ListView* использует *TextView* для вывода на экран значений toString(), принадлежащих элементом массива.
- **Spinner**. Составной элемент, отображающий *TextView* в сочетании с соответствующим *ListView*, которое позволяет выбрать элемент списка для отображения в текстовой строке. Сама строка состоит из объекта *TextView* и кнопки, при нажатии на которую всплывает диалог выбора. Внешне этот элемент напоминает тэг <SELECT> в HTML.
- **Button**. Стандартная кнопка, которую можно нажимать.
- **CheckBox**. Кнопка, имеющая два состояния. Представлена в виде отмеченного или неотмеченного флажка («галки»).

- **RadioButton**. «Радиокнопка», позволяет выбрать только один из нескольких вариантов.
- **ViewFlipper**. Группа представлений (*ViewGroup*), позволяющая определить набор элементов и горизонтальную строку, в которой может выводиться только одно Представление (*View*). При этом переходы между отображающимися элементами осуществляются с помощью анимации.

Android предлагает и более продвинутые реализации Представлений, такие, как элементы для выбора даты и времени, поля ввода с автоматическим дополнением, галереи, вкладки и даже карты (MapView).
Более полный список поддерживаемых системой Представлений можно увидеть по адресу http://developer.android.com/guide/tutorials/views/index.html
Кроме готовых Представлений, разработчик, при необходимости, может создавать собственные, расширяя класс View или его подклассы.

Разметка (Layout)

Разметка (*Layout)* является расширением класса *ViewGroup* и используется для размещения дочерних компонентов на экране устройства. Используя вложенные Разметки, можно создавать пользовательские интерфейсы любой сложности.
Наиболее часто используемые виды Разметки:

- **FrameLayout**. Самая простая разметка, прикрепляет каждое новое дочернее Представление к левому верхнему углу экрана, накладывая новый элемент на предыдущий, заслоняя его.

- **LinearLayout**. Помещает дочерние Представления в горизонтальный или вертикальный ряд. Вертикальная разметка представляет собой колонку, а горизонтальная – строку с элементами. Данная разметка позволяет задавать не только размеры, но и «относительный вес» дочерних элементов, благодаря чему можно гибко контролировать их размещение на экране.

- **RelativeLayout**. Наиболее гибкий среди стандартных видов разметки. Позволяет указывать позиции дочерних Представлений относительно границ свободного пространства и других Представлений.

- **TableLayout**. Позволяет размещать дочерние Представления внутри ячеек «сетки», состоящей из строк и столбцов. Размеры ячеек могут оставаться постоянными или автоматически растягиваться при необходимости.

- **Gallery**. Представляет элементы в виде прокручиваемого горизонтального списка (обычно графические элементы).

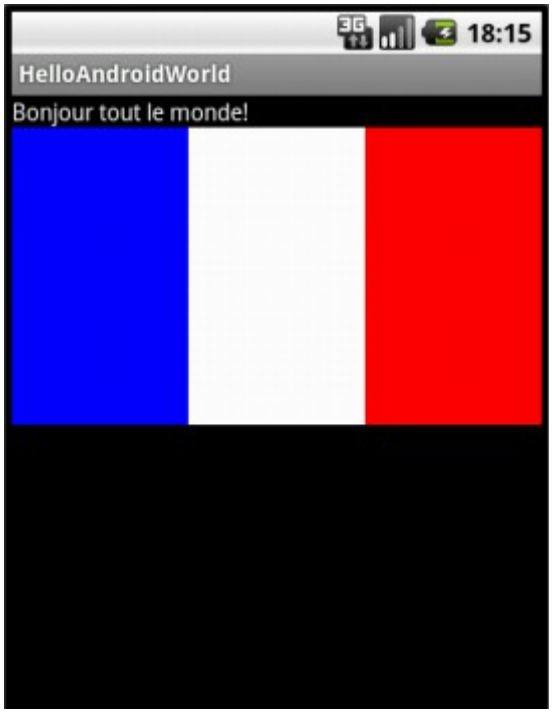
Актуальную информацию о свойствах и возможностям разных видов разметки можно получить по адресу http://developer.android.com/guide/topics/ui/layout-objects.html
Наиболее предпочтительный способ реализации разметки экрана – использование внешних ресурсов: XML-файлов, описывающих размещение элементов на экране и их параметры.
Мы уже работали с содержимым файла **res/layout/main.xml** для изменения внешнего вида приложения HelloAndroidWorld, а теперь подробнее рассмотрим его содержимое:

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

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"
        android:textColor="@color/text_color" />

</LinearLayout>
```

Корневым элементом нашей разметки является <*LinearLayout*>, имеющий один дочерний элемент <*TextView*>. Атрибуты <*LinearLayout*> определяют пространство имен «android»(`xmlns:android="http://schemas.android.com/apk/res/android"`), ширину (`android:layout_width`), высоту (`android:layout_height`) и ориентацию (`android:orientation`), которая определяет способ размещения дочерних элементов внутри *LinearLayout*: вертикально или горизонтально.
Обратите внимание на относительное (относительно размеров родительского элемента), а не абсолютное (в пикселях) задание размеров (ширины и высоты). Такой способ определения размеров является предпочтительным и позволяет создавать дизайн приложений, слабо зависящий от размеров экрана устройства.
Элемент <TextView> в нашем случае имеет следующие атрибуты: ширину и высоту, а также ссылки на строку с именем «*hello*» и цвет, используемый для отображения текста «*text_color*».

Лабораторная работа «Использование LinearLayout»

1. Продолжите локализацию приложения HelloAndroidWorld, теперь для французского языка. Для рисования флага используйте вложенный LinearLayout с вертикальной ориентацией дочерних элементов:



2. Обратите внимание, что при размещении элементов *внутри вложенного LinearLayout* удобно указывать атрибут **android:layout_weight="1"**, в этом случае дочерние виджеты будут размещены по горизонтали равномерно.

3. После получения нужного результата верните стандартные языковые настройки в виртуальном устройстве.

Лабораторная работа «Использование RelativeLayout»

RelativeLayout является очень полезным вариантом разметки, позволяющим создавать пользовательский интерфейс без избыточного количества вложенных элементов *ViewGroup*.

1. Создайте новый проект с именем **RelativeLayoutSample**.
2. Добавьте нужные строки в файл **res/values/strings.xml** и удалите строку с именем **hello**:

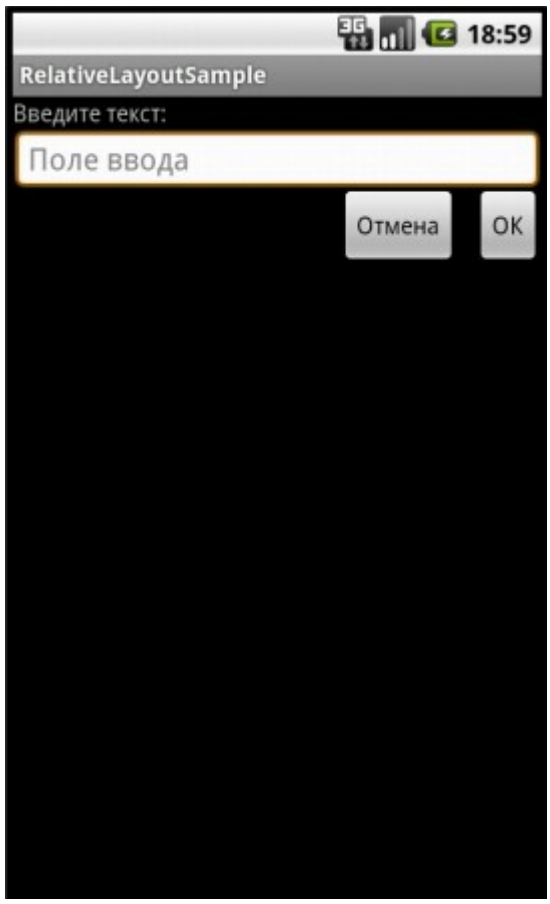
```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="label_text">Введите текст:</string>
    <string name="entry_hint">Поле ввода</string>
```

```
    <string name="app_name">RelativeLayoutSample</string>

</resources>
```

3. Отредактируйте файл разметки **res/layout/main.xml** так, чтобы он имел следующее содержимое:

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

    <TextView
        android:id="@+id/label"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/label_text" />

    <EditText
        android:id="@+id/entry"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/label"
        android:background="@android:drawable/editbox_background"
        android:hint="@string/entry_hint" />

    <Button
        android:id="@+id/ok"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_below="@id/entry"
        android:layout_marginLeft="10dip"
        android:text="@android:string/ok" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignTop="@id/ok"
        android:layout_toLeftOf="@id/ok"
        android:text="@android:string/cancel" />

</RelativeLayout>
```

4. Запустите приложение:



5. Отредактируйте файл **AndroidManifest.xml**, чтобы изменить тему, используемую приложением. Для это в узел **<application>** внесите следующие изменения:

```
. . .
<application
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.Light" >
. . .
```

6. Запустите приложение с новой темой:

7. Мы использовали для текста кнопок в разметке стандартные строковые значения, которые предоставляет Android. Поэкспериментируйте с настройками языка в виртуальном устройстве и обратите внимание, как при запуске приложения меняются надписи на *кнопках*. Очевидно, что использование стандартных строковых значений позволяет минимизировать затраты на локализацию приложений.

Лабораторная работа «Использование WebView»

В данной лабораторной работе рассматривается использование виджета web-браузера и применяется итеративный подход к созданию приложения.

1. Создайте новый проект с именем **WebViewSample**.
2. Отредактируйте файл **res/layout/main.xml**:

```
<?xml version="1.0" encoding="utf-8"?>
<WebView
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/webview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" />
```

3. Добавьте в конец метода **onCreate** Активности

WebViewSampleActivity следующие строки:

```
WebView webView = (WebView) findViewById(R.id.webview);
webView.getSettings().setJavaScriptEnabled(true);
webView.loadUrl("http://www.ya.ru");
```

4. Запустите приложение:

5. Очевидно, что ~~в супе чего-то не хватает~~ у приложения отсутствуют полномочия на доступ к сети. Добавим нужные полномочия в Манифест приложения, добавив элемент <*uses-permission*> внутри корневого узла <*mainfest*>:

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

6. Запустим проект (нижняя часть картинки отрезана):

7. Продолжим улучшать приложение. Для увеличения полезной площади экрана запретим показ заголовка, для этого укажем соответствующую тему в файле Манифеста:

```
<activity
      android:name=".WebViewSampleActivity"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.NoTitleBar" >
```

8. Запустим проект и убедимся том, что (первая) цель достигнута.
9. В настоящий момент ссылки, ведущие за пределы сайта www.ya.ru обслуживаются стандартным web-браузером, а не нашим WebView, так как оно не в состоянии обработать эти запросы и виджет webView автоматически посылает системе соответствующее *Намерение* (*Intent*), обрабатываемое стандартным браузером. У этой проблемы есть два решения:
   - Добавить в Манифест нужный *Фильтр Намерений* (*Intent Filter*)
   - Переопределить внутри нашей Активности класс *WebViewClient*, чтобы приложение могло обрабатывать свои *собственные запросы* на отображение web-ресурсов с помощью имеющегося виджета WebView.

   Второй вариант является более приемлемым еще и потому, что в этом случае не будет рассматриваться системой как альтернативный web-браузер, что произошло бы в случае добавления *Фильтра Намерений* в Манифест.
10. Вынесем объект *webView* из метода *onCreate* и сделаем его членом класса, после чего добавим внутри Активности новый класс *WebViewSampleClient*, расширяющий класс *WebViewClient* , после чего установим свой обработчик запросов на отображение web-ресурсов:

```
public class WebViewSampleActivity extends Activity {
```

```
    WebView webView;

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

        webView = (WebView) findViewById(R.id.webview);
        webView.getSettings().setJavaScriptEnabled(true);
        webView.loadUrl("http://www.ya.ru");

        webView.setWebViewClient(new WebViewSampleClient());
    }

    private class WebViewSampleClient extends WebViewClient {
        @Override
        public boolean shouldOverrideUrlLoading(WebView view,
              String url) {
            view.loadUrl(url);
            return true;
        }
    }
}
```

11. Запустим приложение и убедимся, что
12. Остался серьезный недостаток – «неправильная» обработка нажатия кнопки «назад» устройства, приложение при этом заканчивает свою работу. Решение этой проблемы простое и прямолинейное: сделаем свой обработчик событий нажатия на кнопки, в котором будет реализовано только одно действие: если была нажата кнопка «назад», виджет WebView получит команду вернуться на предыдущую страницу (если у него есть такая возможность). Переопределим метод **onKeyDown** в Активности:

```
    @Override
    public boolean onKeyDown(int keyCode, KeyEvent event) {
        if ((keyCode == KeyEvent.KEYCODE_BACK) &&
              webView.canGoBack()) {
            webView.goBack();
            return true;
        }
        return super.onKeyDown(keyCode, event);
    }
```

13. Запустим приложение и убедимся, что цель достигнута.

Адаптеры в Android

Адаптеры в Android являются связующими классами между данными приложения и Представлениями. Адаптер отвечает за создание дочерних Представлений, отображающими каждый элемент внутри родительского виджета, а также обеспечивает доступ к исходным данным, используемым приложением. Представления, использующие привязку к Адаптеру, должны быть потомками абстрактного класса *AdapterView*.
Android содержит набор стандартных Адаптеров, которые доставляют данные в стандартные виджеты пользовательского интерфейса. Двумя наиболее полезными и часто используемыми Адаптерами являются *ArrayAdapter* и *SimpleCursorAdapter*.
ArrayAdapter использует механизм обобщенных типов (*generics*) языка Java для привязки родительского класса *AdapterView* к массиву объектов указанного типа. По умолчанию *ArrayAdapter* использует метод *toString()* для каждого элемента в массиве, чтобы создать и заполнить текстовыми данными виджеты *TextView*.
SimpleCursorAdapter привязывает указанное в разметке Представление к столбцам Курсора, ставшего результатом запроса к СУБД или Контент-Провайдеру. Для его использования требуется описать разметку в формате XML, а затем привязать каждый столбец к Представлениям из этой разметки. Адаптер создаст Представления для каждой записи из Курсора и наполнит их данными из соответствующих столбцов.
SimpleAdapter позволяет привязать *ListView* к списку *ArrayList*, содержащему объекты типа *Map* (ассоциативные массивы, содержащие пары «ключ-значение»). Для каждого такого объекта при отображении используется один элемент из *ListView*. Как и для *SimpleCursorAdapter*, для отображения применяется XML-разметка, к элементам которой привязываются члены каждого объекта типа Map.

Использование Адаптеров для привязки данных.

Чтобы применить Адаптер, необходимо вызвать из Представления (потомка абстрактного класса *AdapterView*) метод *setAdapter()* (или его более конкретизированный вариант). Пример использования *ArrayAdapter*:

```
ArrayList<String> myStringArray = new ArrayList<String>();
ArrayAdapter<String> myAdapterInstance;

int layoutID = android.R.layout.simple_list_item_1;
myAdapterInstance = new ArrayAdapter<String>(this, layoutID,
          myStringArray);

myListView.setAdapter(myAdapterInstance);
```

Пример использования *SimpleCursorAdapter*:

```
String uriString = "content://contacts/people/";
Cursor myCursor = managedQuery(Uri.parse(uriString), null,
          null, null, null);

String[] fromColumns = new String[] {People.NUMBER,
     People.NAME};
int[] toLayoutIDs = new int[] {R.id.numberTextView,
     R.id.nameTextView};

SimpleCursorAdapter myAdapter;

myAdapter = new SimpleCursorAdapter(this,
     R.layout.simplecursorlayout,
          myCursor, fromColumns, toLayoutIDs);

myListView.setAdapter(myAdapter);
```

Лабораторная работа «Использование ListView»

1. Создайте новый Android проект **ListViewSample**.
2. В каталоге **res/values** создайте файл **arrays.xml** со следующим содержимым:

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

    <string-array name="stations">
        <item>Авиамоторная</item>
        <item>Автозаводская</item>
        <item>Академическая</item>
        <item>Александровский сад</item>
        <item>Алексеевская</item>
        <item>Алтуфьево</item>
        <item>Аннино</item>
        <item>Арбатская (Арбатско-Покровская линия)</item>
        <item>Арбатская (Филевская линия)</item>
        <item>Аэропорт</item>
        <item>Бабушкинская</item>
        <item>Багратионовская</item>
        <item>Баррикадная</item>
        <item>Бауманская</item>
        <item>Беговая</item>
        <item>Белорусская</item>
        <item>Беляево</item>
        <item>Бибирево</item>
        <item>Библиотека имени Ленина</item>
        <item>Битцевский парк</item>
```

```
<item>Борисовская</item>
<item>Боровицкая</item>
<item>Ботанический сад</item>
<item>Братиславская</item>
<item>Бульвар адмирала Ушакова</item>
<item>Бульвар Дмитрия Донского</item>
<item>Бунинская аллея</item>
<item>Варшавская</item>
<item>ВДНХ</item>
<item>Владыкино</item>
<item>Водный стадион</item>
<item>Войковская</item>
<item>Волгоградский проспект</item>
<item>Волжская</item>
<item>Волоколамская</item>
<item>Воробьевы горы</item>
<item>Выставочная</item>
<item>Выхино</item>
<item>Деловой центр</item>
<item>Динамо</item>
<item>Дмитровская</item>
<item>Добрынинская</item>
<item>Домодедовская</item>
<item>Достоевская</item>
<item>Дубровка</item>
<item>Жулебино</item>
<item>Зябликово</item>
<item>Измайловская</item>
<item>Калужская</item>
<item>Кантемировская</item>
<item>Каховская</item>
<item>Каширская</item>
<item>Киевская</item>
<item>Китай-город</item>
<item>Кожуховская</item>
<item>Коломенская</item>
<item>Комсомольская</item>
<item>Коньково</item>
<item>Красногвардейская</item>
<item>Краснопресненская</item>
<item>Красносельская</item>
<item>Красные ворота</item>
<item>Крестьянская застава</item>
<item>Кропоткинская</item>
<item>Крылатское</item>
<item>Кузнецкий мост</item>
<item>Кузьминки</item>
<item>Кунцевская</item>
<item>Курская</item>
<item>Кутузовская</item>
<item>Ленинский проспект</item>
<item>Лубянка</item>
<item>Люблино</item>
<item>Марксистская</item>
<item>Марьина роща</item>
<item>Марьино</item>
<item>Маяковская</item>
<item>Медведково</item>
<item>Международная</item>
<item>Менделеевская</item>
```

```
<item>Митино</item>
<item>Молодежная</item>
<item>Мякинино</item>
<item>Нагатинская</item>
<item>Нагорная</item>
<item>Нахимовский проспект</item>
<item>Новогиреево</item>
<item>Новокузнецкая</item>
<item>Новослободская</item>
<item>Новоясеневская</item>
<item>Новые Черемушки</item>
<item>Октябрьская</item>
<item>Октябрьское поле</item>
<item>Орехово</item>
<item>Отрадное</item>
<item>Охотныйряд</item>
<item>Павелецкая</item>
<item>Парк культуры</item>
<item>Парк Победы</item>
<item>Партизанская</item>
<item>Первомайская</item>
<item>Перово</item>
<item>Петровско-Разумовская</item>
<item>Печатники</item>
<item>Пионерская</item>
<item>Планерная</item>
<item>Площадь Ильича</item>
<item>Площадь Революции</item>
<item>Полежаевская</item>
<item>Полянка</item>
<item>Пражская</item>
<item>Преображенская площадь</item>
<item>Пролетарская</item>
<item>Проспект Вернадского</item>
<item>Проспект Мира</item>
<item>Профсоюзная</item>
<item>Пушкинская</item>
<item>Речной вокзал</item>
<item>Рижская</item>
<item>Римская</item>
<item>Рязанский проспект</item>
<item>Савеловская</item>
<item>Свиблово</item>
<item>Севастопольская</item>
<item>Семеновская</item>
<item>Серпуховская</item>
<item>Славянский бульвар</item>
<item>Смоленская (Арбатско-Покровская линия)</item>
<item>Смоленская (Филевская линия)</item>
<item>Сокол</item>
<item>Сокольники</item>
<item>Спортивная</item>
<item>Сретенский бульвар</item>
<item>Строгино</item>
<item>Студенческая</item>
<item>Сухаревская</item>
<item>Сходненская</item>
<item>Таганская</item>
<item>Тверская</item>
<item>Театральная</item>
```

```
        <item>Текстильщики</item>
        <item>Теплый Стан</item>
        <item>Тимирязевская</item>
        <item>Третьяковская</item>
        <item>Трубная</item>
        <item>Тульская</item>
        <item>Тургеневская</item>
        <item>Тушинская</item>
        <item>Улица 1905года</item>
        <item>Улица Академика Янгеля</item>
        <item>Улица Горчакова</item>
        <item>Улица Подбельского</item>
        <item>Улица Скобелевская</item>
        <item>Улица Старокачаловская</item>
        <item>Университет</item>
        <item>Филевский парк</item>
        <item>Фили</item>
        <item>Фрунзенская</item>
        <item>Царицыно</item>
        <item>Цветной бульвар</item>
        <item>Черкизовская</item>
        <item>Чертановская</item>
    </string-array>

</resources>
```

3. В каталоге res/layout создайте файл list_item.xml со следующим содержимым:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="10dp"
    android:textSize="16sp" >

</TextView>
```

4. Модифицируйте метод onCreate вашей Активности:

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        Resources r = getResources();
        String[] stationsArray =
                  r.getStringArray(R.array.stations);

        ArrayAdapter<String> aa = new ArrayAdapter<String>(this,
                  R.layout.list_item, stationsArray);

        setListAdapter(aa);
```

```
        ListView lv = getListView();

    }
```

5. Измените базовый класс Активности с **Activity** на **ListActivity**.
6. Запустите приложение.
7. Для реакции на клики по элементам списка требуется добавить обработчик такого события, с помощью метода *setOnItemClickListener*. В качестве обработчика будет использоваться анонимный объект класса *OnItemClickListener*. Добавьте следующий код в *нужное место*:
8.

```
        lv.setOnItemClickListener(new OnItemClickListener() {
            public void onItemClick(AdapterView<?> parent,
                    View v, int position, long id) {

                CharSequence text = ((TextView) v).getText();
                int duration = Toast.LENGTH_LONG;
                Context context = getApplicationContext();
                Toast.makeText(context, text, duration).show();
            }
        });
```

9. Запустите приложение и «покликайте» по станциям метро.

Лабораторная работа «Использование управляющих элементов в пользовательском интерфейсе»

Цель лабораторной работы – научиться использовать в интерфейсе пользователя различные управляющие элементы: кнопки с изображениями, радиокнопки, чекбоксы и пр.

Подготовка

1. Создайте новый проект **ControlsSample**.
2. Отредактируйте файл **res/layout/main.xml** так, чтобы остался только корневой элемент **LinearLayout**. В него в дальнейшем будут добавляться необходимые дочерние элементы:

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

</LinearLayout>
```

Использование графической кнопки

Для использования изображения вместо текста на кнопке потребуются три изображения для трех состояний кнопки: обычное, выбранное («в фокусе») и нажатое. Все эти три изображения с соответствующими состояниями описываются в одном XML файле, который используется для создания такой кнопки.

1. Скопируйте нужные изображения кнопки в каталог **res/drawable-mdpi**, для обновления списка содержимого каталога в **Eclipse** можно использовать кнопку **F5**.
2. В этом же каталоге создайте файл smile_button.xml, описывающий, какие изображения в каких состояниях кнопки нужно использовать:

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

    <item android:drawable="@drawable/smile_pressed"
     android:state_pressed="true"/>
    <item android:drawable="@drawable/smile_focused"
     android:state_focused="true"/>
    <item android:drawable="@drawable/smile_normal" />

</selector>
```

3. Добавьте элемент **Button** внутри **LinearLayout** в файле разметки **res/layout/main.xml**:

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="@drawable/smile_button"
    android:onClick="onButtonClicked"
    android:padding="10dp" />
```

4. Обратите внимание на атрибут `android:onClick="onButtonClicked"`, указывающий, какой метод из Активности будет использоваться как обработчик нажатия на данную кнопку. Добавьте этот метод в Активность:

```
public void onButtonClicked(View v) {
    Toast.makeText(this, "Кнопка нажата", Toast.LENGTH_SHORT)
            .show();
}
```

5. Запустите приложение и посмотрите, как изменяется изображение кнопки в разных состояниях, а также как функционирует обработчик

нажатия на кнопку.

Использование виджета CheckBox

1. Добавьте элемент **CheckBox** внутри **LinearLayout** в файле разметки **res/layout/main.xml**:
2.

```
<CheckBox
    android:id="@+id/checkbox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="onCheckboxClicked"
    android:text="Выбери меня" />
```

3. Атрибут `android:onClick="onCheckboxClicked"` определяет, какой метод из Активности будет использоваться как обработчик нажатия на виджет. Добавьте этот метод в Активность:

```
public void onCheckboxClicked(View v) {
    if (((CheckBox) v).isChecked()) {
        Toast.makeText(this, "Отмечено", Toast.LENGTH_SHORT)
           .show();
    } else {
        Toast.makeText(this, "Не отмечено", Toast
           .LENGTH_SHORT).show();
    }
}
```

4. Запустите приложение и посмотрите на поведение чекбокса в разных ситуациях.

Использование виджета ToggleButton

Данный виджет хорошо подходит в качестве альтернативы радиокнопкам и чекбоксам, когда требуется переключаться между двумя взаимоисключающими состояниями, например, *Включено/Выключено*.

1. Добавьте элемент **ToggleButton** внутри **LinearLayout** в файле разметки **res/layout/main.xml**:

```
<ToggleButton android:id="@+id/togglebutton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOn="Звонок включен"
    android:textOff="Звонок выключен"
    android:onClick="onToggleClicked"/>
```

2. Атрибут `android:onClick="onToggleClicked"` определяет,

какой метод из Активности будет использоваться как обработчик нажатия на виджет. Добавьте этот метод в Активность:

```
public void onToggleClicked(View v) {
    if (((ToggleButton) v).isChecked()) {
        Toast.makeText(this, "Включено", Toast
           .LENGTH_SHORT).show();
    } else {
        Toast.makeText(this, "Выключено", Toast
           .LENGTH_SHORT).show();
    }
}
```

3. Запустите приложение и проверьте его функционирование.

Использование виджета RadioButton

Радиокнопки используются для выбора между различными взаимоисключающими вариантами. Для создания *группы радиокнопок* используется элемент *RadioGroup*, внутри которого располагаются элементы *RadioButton*.

1. Добавьте следующие элементы разметки внутри **LinearLayout** в файле **res/layout/main.xml**:

```
<RadioGroup
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical" >

    <RadioButton
        android:id="@+id/radio_dog"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="onRadioButtonClicked"
        android:text="Собачка" />

    <RadioButton
        android:id="@+id/radio_cat"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="onRadioButtonClicked"
        android:text="Кошечка" />

    <RadioButton
        android:id="@+id/radio_rabbit"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="onRadioButtonClicked"
```

```
        android:text="Кролик" />
</RadioGroup>
```

2. Добавьте метод **onRadioButtonClicked** в Активность:

```
public void onRadioButtonClicked(View v) {
     RadioButton rb = (RadioButton) v;
     Toast.makeText(this, "Выбрано животное: "
          + rb.getText(), Toast.LENGTH_SHORT).show();
}
```

3. Проверьте работу приложения.

Использование виджета EditText

Виджет *EditText* используется для ввода текста пользователем. Установленный для этого виджета обработчик нажатий на кнопки будет показывать введенный текст с помощью *Toast*.

1. Добавьте элемент **EditText** внутри **LinearLayout** в файле разметки **res/layout/main.xml**:

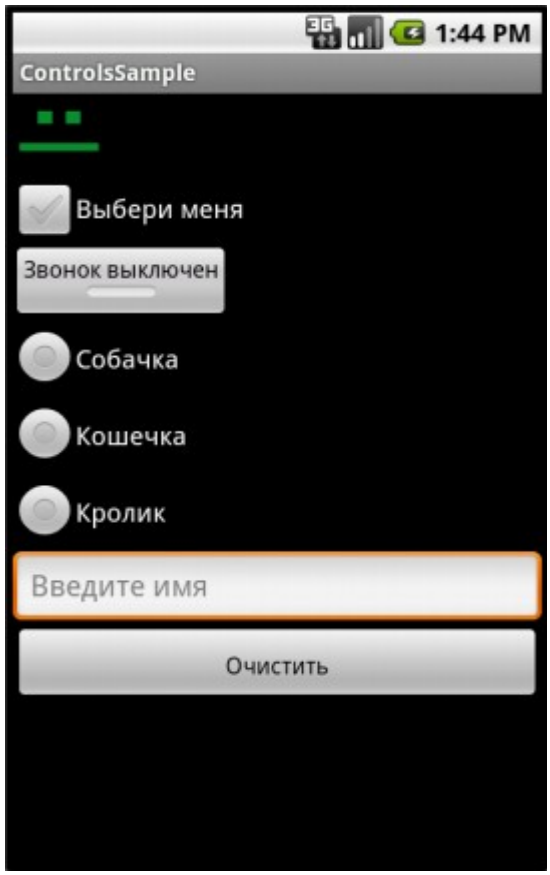
```
<EditText
    android:id="@+id/user_name"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="Введите имя"/>
```

2. Для обработки введенного пользователем текста добавьте следующий код в конце метода **onCreate**. Обратите внимание, этот обработчик, в отличие от предыдущих, использованных нами, *возвращает* значение **true** или **false**. Семантика этих значений традиционна: *true* означает, что событие (*event*) обработано и больше никаких действий не требуется, *false* означает, что событие *не обработано этим обработчиком* и будет передано следующим обработчикам в цепочке. В нашем случае реагирование происходит только на нажатие (*ACTION_DOWN*) кнопки Enter (*KEYCODE_ENTER*):

```
final EditText userName = (EditText) findViewById(
     R.id.user_name);

userName.setOnKeyListener(new View.OnKeyListener() {

     @Override
     public boolean onKey(View v, int keyCode, KeyEvent
          event) {
          if ((event.getAction() == KeyEvent.ACTION_DOWN)
                    && (keyCode == KeyEvent.KEYCODE_ENTER)) {
```

```
                Toast.makeText(getApplicationContext(),
                        userName.getText(), Toast.LENGTH_SHORT)
                        .show();

                return true;
            }
            return false;
        }

});
```

3. Запустите приложение и проверьте его работу.
4. Добавьте кнопку **«Очистить»** в разметку и напишите обработчик, очищающий текстовое поле (используйте метод *setText* виджета *EditText*):



5. Проверьте работу приложения.

Намерения в Android

Намерения (Intent) в Android используются в качестве механизма передачи сообщений, который может работать как внутри одного приложения, так и между приложениями. *Намерения* могут применяться для:

- объявления о желании (необходимости) вашего приложения запуска какой-то Активности или Сервиса для выполнения определенных действий.
- Извещения о том, что произошло какое-то событие.
- Явного запуска указанного Сервиса или Активности.

Последний вариант является наиболее часто используемым.

Использование Намерений для запуска Активностей.

Чтобы запустить нужную Активность, вызывается метод *startActivity(someIntent).*
В конструкторе Намерения можно явно указать *класс Активности*, которую требуется запустить, или действие, которое нужно выполнить. Во втором случае система автоматически подберет нужную Активность, используя механизм, называемый *Определением Намерений (Intent Resolution)*. Метод *startActivity* находит запускает Активность, наиболее подходящую вашему Намерению.
По окончании работы запущенной таким образом Активности *запустившая* ее Активность не получает никаких извещений о результатах обработки Намерения. Если требуется получать результаты, используется метод *startActivityForResult*.
Для явного указания того, какую Активность (конкретный класс в приложении) требуется запустить, создаются Намерения с помощью указания параметров следующих конструктора: *текущий Контекст* приложения и *класс Активности* для запуска.

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

Возвращение результатов работы Активности

Активность, запущенная с помощью метода *startActivity*, полностью независима от запустившей ее Активности и, соответственно, завершает свою работу, никому об этом не сообщая. В то же время, вы можете запускать Активности, «связанные» со своим «породителем». Такой способ отлично подходит для ситуаций, когда «дочерняя» Активность должна обработать

ввод пользовательских данных и предоставить результаты обработки «родительской» Активности. Запускаемые таким образом (с помощью метода *startActivityForResult*) Активности должны быть «зарегистрированы» в Манифесте приложения.
В отличие от метода *startActivity*, метод s*tartActivityForResult* требует явного указания еще одного параметра – *кода запроса (request code)*. Этот параметр используется *вызывающей* Активностью для определения того, какая именно дочерняя Активность завершила работу и (возможно) предоставила результаты:

```
private static final int BUY_BEER = 1;

Intent intent = new Intent(this, MyOtherActivity.class);
startActivityForResult(intent, BUY_BEER);
```

Возвращение результатов работы

Когда дочерняя Активность готова к завершению работы, до вызова метода *finish* требуется вызвать метод *setResult* для передачи результатов вызывающей Активности.
Метод *setResult* получает два параметра: *код возврата* и сам *результат*, представленный в виде Намерения.
Код возврата (result code), возвращаемый из дочерней Активности – это, обычно, либо *Activity.RESULT_OK*, либо *Activity.RESULT_CANCELED*. В случае, если дочерняя Активность завершит работу без вызова метода setResult, код возврата, переданный родительской Активности, будет равен *Activity.RESULT_CANCELED*.
В некоторых случаях может потребоваться использовать собственные коды возврата для обработки определенных ситуаций. Метод *setResult* в качестве *кода возврата* воспринимает любое целочисленное значение.
Намерение, возвращаемое как *результат*, часто содержит URI, указывающий на конкретный объект данных, и/или набор дополнительных значений, записанных в *свойство* Намерения *extras*, использующихся для передачи дополнительной информации.
Пример указания обработчика нажатия на кнопку button, устанавливающего *результат работы* и завершающего текущую Активность:

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

     public void onClick(View v) {
          Intent result = new Intent();
          result.putExtra("teacher name", "Mike Varakin");

          setResult(RESULT_OK, result);
```

```
                finish();
            }
        });
```

Обработка результатов дочерней Активности

Когда дочерняя Активность завершает работу, в родительской Активности вызывается обработчик *onActivityResult*, который получает следующие параметры:

- **Request code**. Использованный при запуске дочерней активности *код запроса*.
- **Result code**. Код возврата.
- **Data**. Намерение, используемое для упаковки возвращаемых данных.

Пример обработчика *onActivityResult*:

```
private static final int SELECT_VICTIM = 1;
private static final int DRINK_BEER = 2;

@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {

     switch (requestCode) {
          case (SELECT_VICTIM): {
               if (resultCode == Activity.RESULT_OK) {
                    String selectedVictim = data.
                         getStringExtra("victim");
               }
               break;
          }

          case (DRINK_BEER): {
               if (resultCode == Activity.RESULT_OK) {
                    // Обработать результаты заказанного действия
               }
               break;

          default:
               super.onActivityResult(requestCode, resultCode,
                    data);
          }
     }
}
```

Лабораторная работа «Вызов Активности с помощью явного намерения и получение результатов работы»

1. Создайте новый проект **MetroPicker**.
2. Добавьте вспомогательную Активность **ListViewActivity** для отображения и выбора станций метро, в качестве заготовки используйте результаты лабораторной работы **«Использование ListView»** .
3. Отредактируйте файл разметки **res/layout/main.xml**: добавьте кнопку выбора станции метро, присвоив идентификаторы виджетам *TextView* и *Button* для того, чтобы на них можно было ссылаться в коде.
4. Установите обработчик нажатия на кнопку в главной Активности для вызова списка станции и выбора нужной станции.
5. Напишите нужный обработчик для установки выбранной станции метро в виджет *TextView* родительской Активности (метод *setText* виджета *TextView* позволяет установить отображаемый текст). Не забудьте обработать ситуацию, когда пользователь нажимает кнопку «Назад» (в этом случае «никакой станции не выбрано» и главная Активность должна известить об этом пользователя).
6. Убедитесь в работоспособности созданного приложения, проверив реакцию различные действия потенциальных пользователей.

Неявные намерения

Неявные намерения используются для запуска Активностей для выполнения заказанных действий в условиях, когда неизвестно (или безразлично), какая именно Активность (и из какого приложения) будет использоваться.
При создании Намерения, которое в дальнейшем будет передано методу *startActivity*, необходимо назначить действие (*action*), которое нужно выполнить, и, возможно, указать *URI данных*, которые нужно обработать. Также можно передать дополнительную информацию с помощью свойства *extras* Намерения. Android сам найдет подходящую Активность (основываясь на характеристиках Намерения) и запустит ее. Пример неявного вызова телефонной «звонилки»:

```
Intent intent = new Intent(Intent.ACTION_DIAL,
          Uri.parse("tel:(495)502-99-11"));
startActivity(intent);
```

Для определения того, какой именно компонент должен быть запущен для выполнения действий, указанных в Намерениях, Android использует *Фильтры Намерений* (*Intent Filters*). Используя *Фильтры Намерений*, приложения сообщают системе, что они могут выполнять определенные *действия* (*action*) с определенными данными (*data*) при определенных условиях (*category*) по заказу других компонентов системы.

Для регистрации компонента приложения (Активности или Сервиса) в качестве потенциального обработчика Намерений, требуется добавить элемент **<intent-filter>** в качестве дочернего элемента для нужного компонента в Манифесте приложения. У элемента **<intent-filter>** могут быть указаны следующие дочерние элементы (и соответствующие атрибуты у них):

- **<action>**. Атрибут *android:name* данного элемента используется для указания названия действия, которое может обслуживаться. Каждый Фильтр Намерений должен содержать не менее одного вложенного элемента *<action>*. Если не указать *действие*, ни одно Намерение не будет «проходить» через этот Фильтр. У главной Активности приложения в Манифесте должен быть указан Фильтр Намерений с *действием* ***android.intent.action.MAIN***
- **<category>**. Сообщает системе, при каких обстоятельствах должно обслуживаться *действие* (с помощью атрибута *android:name)*. Внутри *<intent-filter>* может быть указано несколько категорий. Категория ***android.intent.category.LAUNCHER*** требуется Активности, которая желает иметь «иконку» для запуска. Активности, запускаемые с помощью метода startActivity, обязаны иметь категорию ***android.intent.category.DEFAULT***
- **<data>**. Дает возможность указать тип данных, которые может обрабатывать компонент. *<intent-filter>* может содержать несколько элементов *<data>*. В этом элементе могут использоваться следующие атрибуты:
  - *android:host* : имя хоста (например, *www.specialist.ru*)
  - *android:mimetype* : обрабатываемый тип данных (например, *text/html*)
  - *android:path* : «путь» внутри URI (например, */course/android*)
  - *android:port* : порт сервера (например, *80*)
  - *android:scheme* : схема URI (например, *http*)

Пример указания Фильтра Намерений:

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

    <intent-filter>

      <action android:name="android.intent.action.SEND" />

      <category android:name=
     "android.intent.category.DEFAULT" />

      <data android:mimeType="text/plain" />
```

```
        </intent-filter>

    </activity>
```

При запуске Активности с помощью метода *startActivity* неявное Намерение обычно подходит только одной Активности. Если для данного Намерения подходят несколько Активностей, пользователю предлагается список вариантов.
Определение того, какие Активности подходят для Намерения, называется *Intent Resolution*. Его задача – определить наиболее подходящие Фильтры Намерений, принадлежащие компонентам установленных приложений. Для этого используются следующие проверки в указанном порядке:

- **Проверка действий**. После этого шага остаются только компоненты приложений, у которых в Фильтрах Намерений указано *действие* Намерения. В случае, если *действие* в Намерении отсутствует, совпадение происходит для всех Фильтров Намерений, у которых указано хотя бы одно *действие*.
- **Проверка категорий**. Все категории, имеющиеся у Намерения, должны присутствовать в Фильтре Намерений. Если у Намерения нет категорий, то на данном этапе ему соответствуют все Фильтры Намерений, за одним исключением, упоминавшимся выше: Активности, запускаемые с помощью метода startActivity, обязаны иметь категорию ***android.intent.category.DEFAULT***, так как Намерению, использованному в этом случае, по умолчанию присваивается данная категория, даже если разработчик не указал ничего явно. Из этого исключения, в свою очередь, есть исключение: если у Активности присутствуют действие ***android.intent.action.MAIN*** и категория ***android.intent.category.LAUNCHER***, ему не требуется иметь категорию ***android.intent.category.DEFAULT***.
- **Проверка данных**. Здесь применяются следующие правила:
  - Намерение, не содержащее ни *URI*, ни *типа данных*, проходит через Фильтр, если он тоже ничего перечисленного не содержит.
  - Намерение, которое имеет *URI*, но не содержит *тип данных* (и *тип данных* невозможно определить по *URI*), проходит через Фильтр, если *URI* Намерения совпадает с *URI* Фильтра. Это справедливо только в случае таких URI, как *mailto:* или *tel:*, которые не ссылаются на реальные данные.
  - Намерение, содержащее *тип данных*, но не содержащее *URI* подходит только для аналогичных Фильтров Намерений.
  - Намерение, содержащее и *тип данных*, и *URI* (или если тип данных может быть вычислен из *URI*), проходит этот этап проверки, только

если его *тип данных* присутствует в Фильтре. В этом случае *URI* должен совпадать, либо(!) у Намерения указан URI вида *content:* или *file:*, а у Фильтра *URI* не указан. То есть, предполагается, что если у компонента в Фильтре указан только *тип данных*, то он поддерживает URI вида *content:* или *file:*.

В случае, если после всех проверок остается несколько приложений, пользователю предлагается выбрать приложение самому. Если подходящих приложений не найдено, в выпустившей Намерение Активности возникает Исключение.

Лабораторная работа «Использование неявных Намерений»

1. Измените проект **MetroPicker** так, чтобы для запуска Активности **ListViewActivity** использовалось неявное Намерение с действием (*action*) , определенным в вашем приложении и имеющем значение "**`com.example.metropicker.intent.action.PICK_METRO_STATION`**".
2. Проверьте работу приложения.

Определение Намерения, вызвавшего запуск Активности

Поскольку объекты типа *Intent* служат, в том числе, для передачи информации между компонентами одного или нескольких приложений, может возникнуть необходимость в работе с объектом Намерения, вызвавшим Активность к жизни.

Для получения доступа к этому объекту используется метод *getIntent*.

Пример:

```
Intent intent = getIntent();
```

Имея данный объект, можно получить доступ к информации, содержащейся в нем:

- метод *getAction* возвращает действие Намерения
- метод *getData* возвращает данные Намерения (обычно *URI*)
- набор методов для разных типов вида *get**ТИП**Extra* позволяет получить доступ к типизированным значениям, хранящимся в свойстве *extras* Намерения.

Примеры:

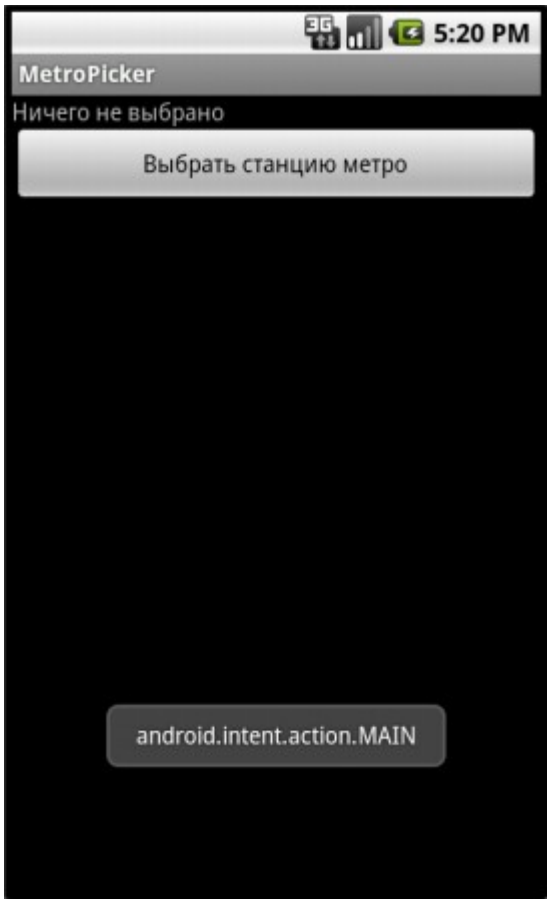
```
String action = intent.getAction();
Uri data = intent.getData();
```

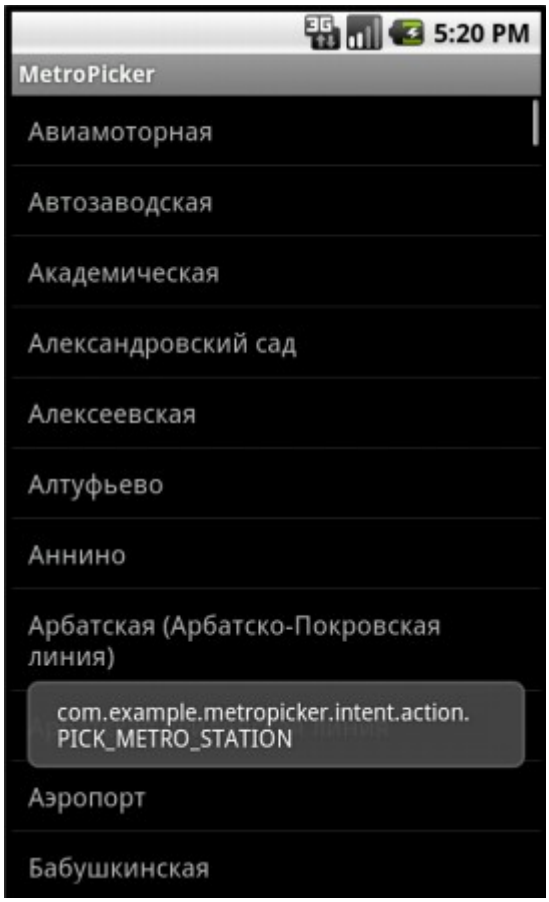
Лабораторная работа «Получение данных из Намерения»

1. Модифицируйте методы onCreate двух ваших Активностей из

предыдущей лабораторной работы так, чтобы с помощью Toast они показывали действие вызвавшего их Намерения .

2. Проверьте работу приложения:





Сохранение состояния и настроек приложения

Поскольку жизненный цикл приложений и Активностей в Android может прерваться в любой момент, для повышения привлекательности и удобства пользовательского интерфейса желательно иметь возможность сохранять (и восстанавливать!) состояния Активностей не только при выходе из активного состояния, но и между запусками. Android предлагает для этого следующие механизмы:

- **Общие Настройки (Shared preferences)**: простой способ сохранения состояния UI и настроек приложения, использующий *пары ключ/значение* для хранения примитивных типов данных и обеспечения доступа к ним по имени.
- **Сохранение состояния приложения**: Для Активностей существуют обработчики событий, позволяющие сохранять и восстанавливать текущее состояние UI, когда выходит и возвращается в активный режим. Для сохранения состояния используется объект класса *Bundle*, который передается методам *onSaveInstanceState* (для сохранения состояния), *onCreate* и *onRestoreInstanceState* (для восстановления). Для доступа к данным в этом случае также используются пары ключ/значение. Обработчики событий из *суперклассов* берут на себя основную работу по сохранению и восстановлению вида UI, фокуса полей и т. д.
- **Прямая работа с файлами**. Если не подходят описанные выше варианты, приложение может напрямую читать и писать данные из файла. Для этого можно использовать как стандартные классы и методы Java, обеспечивающие ввод/вывод, так и методы *openFileInput* и *openFileOutput*, предоставленные Android, для упрощения чтения и записи потоков, относящихся к локальным файлам.

Общие Настройки (Shared Preferences)

Класс *SharedPreferences* предлагает методы для хранения и получения данных, записываемых в файлы, доступные по умолчанию только конкретному приложению. Такой способ хранения обеспечивает сохранность этих данных не только в течении жизненного цикла приложения, но и между запусками и даже перезагрузкой ОС.

Для сохранения данных в файле используется транзакционный механизм: вначале требуется получить объект класса *SharedPreferences.Editor* для конкретного файла настроек, после чего с помощью методов вида *put**Tun*** этого объекта установить нужные значения. Запись значений производится

методом *commit*. Примеры:
Сохранение данных:

```
private static final String PREFS = "PREFS";
static final String KEY_STATION = "selectedStation";

private SharedPreferences prefs;

private String selectedStation;

prefs = getSharedPreferences(PREFS, MODE_PRIVATE);
Editor editor = prefs.edit();
editor.putString(KEY_STATION, selectedStation);
editor.commit();
```

Получение данных:

```
private static final String NOTHING_SELECTED = "Ничего не выбрано";
prefs = getSharedPreferences(PREFS, MODE_PRIVATE);

selectedStation = prefs.getString(KEY_STATION, NOTHING_SELECTED);
tv.setText(selectedStation);
```

Лабораторная работа «Использование SharedPreferences для сохранения состояния»

1. Модифицируйте методы **onCreate** и **onActivityResult** проекта **MetroPicker** для сохранения выбранной станции метро между запусками приложения.
2. Проверьте работоспособность приложения.

Лабораторная работа «Использование SharedPreferences для сохранения настроек»

1. Модифицируйте проект **ControlsSample** так, чтобы состояние управляющих элементов сохранялось и восстанавливалось между запусками приложения.
2. Проверьте работоспособность приложения.

Работа с файлами

Методы *openFileInput* и *openFileOutput* дают возможность работать только с файлами, находящимися в «персональном» каталоге приложения. Как следствие, указание разделителей («/») в имени файла приведет к выбросу

исключения.
По умолчанию файлы, открытые на запись, перезаписываются. Если это не то, что требуется, при открытии установите режим *MODE_APPEND*.
Пример работы с файлами:

```
String FILE_NAME = "app_data";

// Открытие выходного файлового потока
FileOutputStream fos = openFileOutput(FILE_NAME, Context
     .MODE_PRIVATE);
// Открытие входного файлового потока
FileInputStream fis = openFileInput(FILE_NAME);
```

Через объект *Context* в приложении также возможен доступ к двум полезным методам:

- *fileList* возвращает список файлов приложения
- *deleteFile* удаляет файл из каталога приложения

Использование статических файлов как ресурсов

Если приложению необходимо иметь доступ к информации, которую неудобно хранить, например, в СУБД, эти данные можно использовать в виде «сырых» ресурсов, записав их в файлы в каталоге **res/raw**. Яркий пример подобных данных – словари.
Статические файлы, как следует из названия, доступны только для чтения, для их открытия используется метод **openRawResource**:

```
Resources res = getResources();
InputStream file = res.openRawResource(R.raw.filename);
```

Меню в Android

Использование меню в приложениях позволяет сохранить ценное экранное пространство, которое в противном случае было бы занято относительно редко используемыми элементами пользовательского интерфейса.
Каждая Активность может иметь меню, реализующие специфические только для нее функции. Можно использовать также *контекстные* меню, индивидуальные для каждого Представления на экране.

Основы использования меню

В Android реализована поддержка трехступенчатой системы меню, оптимизированную, в первую очередь, для небольших экранов:

- **Основное меню** возникает внизу на экране при нажатии на кнопку «menu» устройства. Оно может отображать текст и иконки для

ограниченного (по умолчанию, не больше шести) числа пунктов. Для этого меню рекомендуется использовать иконки с цветовой гаммой в виде оттенков серого с элементами рельефности. Это меню не может содержать радиокнопки и чекбоксы. Если число пунктов такого меню превышает максимально допустимое значение, в меню автоматически появляется пункт с надписью «*еще*» («*more*»). При нажатии на него отобразится *Расширенное меню*.

- **Расширенное меню** отображает прокручиваемый список, элементами которого являются пункты, не вошедшие в *основное меню*. В этом списке не могут отображаться иконки, но есть возможность отображения радиокнопок и чекбоксов. Поскольку не существует способа отобразить *расширенное меню* вместо *основного*, об изменении состояния каких-то компонентов приложения или системы рекомендуется уведомлять пользователя с помощью изменения иконок или текста пунктов меню.
- **Дочернее меню** (меню третьего уровня) может быть вызвано из основного или расширенного меню и отображается во всплывающем окне. Вложенность не поддерживается, и попытка вызвать из дочернего еще одно меню приведет к выбросу исключения.

Создание меню

При обращении к меню вызывается метод *onCreateOptionsMenu* Активности и для появления меню на экране его требуется переопределить. Данный метод получает в качестве параметра объект класса *Menu*, который в дальнейшем используется для манипуляций с пунктами меню.
Для добавления новых пунктов в меню используется метод *add* объекта *Menu* со следующими параметрами:

- **Группа**: для объединения пунктов меню для групповой обработки
- **Идентификатор**: уникальный идентификатор пункта меню. Этот идентификатор передается обработчику нажатия на пункт меню – методу *onOptionsItemSelected*.
- **Порядок**: значение, указывающее порядок, в котором пункты меню будут выводиться.
- **Текст**: надпись на данном пункте меню.

После успешного создания меню метод *onCreateOptionsMenu* должен вернуть значение *true*.
Пример показывает создание меню из трех пунктов с использованием строковых ресурсов:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
     super.onCreateOptionsMenu(menu);
```

```
        menu.add(0, Menu.FIRST, Menu.NONE, R.string.menu_item1);
        menu.add(0, Menu.FIRST+1, Menu.NONE,
             R.string.menu_item2);
        menu.add(0, Menu.FIRST+2, Menu.NONE,
             R.string.menu_item3);

        return true;
    }
```

Для поиска пунктов меню по идентификатору можно использовать метод *findItem* объекта *Menu*.

Параметры пунктов меню

Наиболее полезными параметрами пунктов меню являются следующие:

- **Краткие заголовки**: используются в случае, если пункт может отобразиться в основном меню. Устанавливается методом *setTitleCondensed* объекта класса *MenuItem*:
  `menuItem.setTitleCondensed("заголовок");`
- **Иконки**: идентификатор *Drawable*, содержащий нужную картинку:
  `menuItem.setIcon(R.drawable.menu_item_icon);`

- **Обработчик выбора пункта меню**: установить можно, но не рекомендуется из соображений повышения производительности, лучше использовать обработчик всего меню (*onOptionsItemSelected*). Тем не менее, пример:

```
menuItem.setOnMenuItemClickListener(new
     OnMenuItemClickListener() {
     public boolean onMenuItemClick(MenuItem _menuItem) {
               // обработать выбор пункта
          return true;
     }
});
```

- **Намерение**: это намерение автоматически передается методу *startActivity*, если нажатие на пункт меню не было обработано обработчиками *onMenuItemClickListener* и *onOptionsItemSelected*:

  ```
  menuItem.setIntent(new Intent(this,
  MyOtherActivity.class));
  ```

Динамическое изменение пунктов меню

Непосредственно перед выводом меню на экран вызывается метод

onPrepareOptionsMenu текущей Активности, и переопределяя его можно динамически изменять состояние пунктов меню: разрешать/запрещать, делать невидимым, изменять текст и т. д.
Для поиска пункта меню, подлежащего модификации можно использовать метод *findItem* объекта *Menu*, передаваемого в качестве параметра:

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

     super.onPrepareOptionsMenu(menu);
     MenuItem menuItem = menu.findItem(MENU_ITEM);
     //модифицировать пункт меню....
     return true;

}
```

Обработка выбора пункта меню

Android позволяет обрабатывать все пункты меню (выбор их) одним обработчиком *onOptionsItemSelected* . Выбранный пункт меню передается этому обработчику в качестве объекта класса *MenuItem*.
Для реализации нужной реакции на выбор пункта меню требуется определить, что именно было выбрано. Для этого используется метод getItemId переданного в качестве параметра объекта, а полученный результат сравнивается с идентификаторами, использованными при добавлении пунктов в меню в методе *onCreateOptionsMenu*.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
     switch (item.getItemId()) {
          // Проверить каждый известный пункт
          case (MENU_ITEM):
               // сделать что-то...
               return true;
     }
     // Если пункт не обработан, отдаем обработку дальше
     return super.onOptionsItemSelected(item);
}
```

Дочерние и контекстные меню

При появлении на экране дочерние и контекстные меню выглядят одинаково, в виде плавающих окон, но при этом создаются по-разному.

Создание дочерних меню

Для создания дочерних меню используется метод *addSubMenu* объекта класса

Menu:

```
SubMenu sub = menu.addSubMenu(0, 0, Menu.NONE,
     "дочернее меню");
sub.setHeaderIcon(R.drawable.icon);
sub.setIcon(R.drawable.icon);
MenuItem submenuItem = sub.add(0, 0, Menu.NONE,
     "пункт дочернего меню");
```

Как уже было сказано выше, *вложенные* дочерние меню Android *не поддерживает*.

Создание контекстных меню

Наиболее часто используемым способом создания контекстного меню в Android является переопределение метода *onCreateContextMenu* Активности:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
          ContextMenu.ContextMenuInfo menuInfo) {
     super.onCreateContextMenu(menu, v, menuInfo);
     menu.setHeaderTitle("Конекстное меню");
     menu.add(0, Menu.FIRST, Menu.NONE, "Пункт 1");
     menu.add(0, Menu.FIRST+1, Menu.NONE, "Пункт 2");
     menu.add(0, Menu.FIRST+2, Menu.NONE, "Пункт 3");
}
```

Регистрация обработчика контекстного меню для нужных Представлений осуществляется с помощью метода Активности *registerForContextMenu*:

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

     tv = (TextView) findViewById(R.id.text_view);
     registerForContextMenu(tv);
}
```

Пример обработчика:

```
@Override
public boolean onContextItemSelected(MenuItem item) {
     switch (item.getItemId()) {
     case DELETE_ID:
          AdapterContextMenuInfo info =
               (AdapterContextMenuInfo) item
                    .getMenuInfo();
```

```
                db.deleteItem(info.id);
                populate();
                return true;
            }
            return super.onContextItemSelected(item);
        }
```

Описание меню с помощью XML

Часто бывает удобнее всего описывать меню, в том числе иерархические, в виде ресурсов. О преимуществах такого подхода говорилось выше.
Меню традиционно описываются и хранятся в каталоге **res/menu** проекта. Все иерархии меню (если есть иерархические меню) должны находиться в отдельных файлах, а имя файла будет использовать как идентификатор ресурса. Корневым элементом файла должен быть тэг <*menu*>, а пункты меню описываются тэгом <item>. Свойства пунктов меню описываются соответствующими атрибутами:

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

    <item
        android:id="@+id/item01"
        android:icon="@drawable/menu_item"
        android:title="Пункт 1">
    </item>
    <item
        android:id="@+id/item02"
        android:checkable="true"
        android:title="Пункт 2">
    </item>
    <item
        android:id="@+id/item03"
        android:title="Пункт 3">
    </item>
    <item
        android:id="@+id/item04"
        android:title="Дочернее меню 1">
        <menu>
            <item
                android:id="@+id/sub1item01"
                android:title="Пункт дочернего меню 1">
            </item>
        </menu>
    </item>
    <item
        android:id="@+id/item05"
        android:title="Дочернее меню 2">
        <menu>
```

```
            <item
                android:id="@+id/sub2item01"
                android:title="Пункт дочернего меню 2">
            </item>
        </menu>
    </item>

</menu>
```

Для создания объектов Menu из ресурсов в событий *onCreateOptionsMenu* и *onCreateContextMenu* используется метод *inflate* объекта типа *MenuInflater*:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
     super.onCreateOptionsMenu(menu);

     MenuInflater inflater = getMenuInflater();
     inflater.inflate(R.menu.menu1, menu);

     return true;
}
```

или так:

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

     MenuInflater inflater = getMenuInflater();
     inflater.inflate(R.menu.menu2, menu);

     menu.setHeaderTitle("Контекстное меню");
}
```

Лабораторная работа «Создание и использование меню»

Модифицируйте проект **MetroPicker** следующим образом:

1. Добавьте главное меню в Активность, отображающую список станций метро. В меню должен быть один пункт: «вернуться». Меню создайте динамически в коде, без использования строковых ресурсов.
2. Динамически создайте контекстное меню для Представления *TextView*, отображающего выбранную станцию метро главной Активности. Выбор пункта меню должен сбрасывать выбранную станцию.
3. Для главной Активности создайте основное меню из двух пунктов: «сбросить» и «выйти». Реализуйте нужные функции при выборе этих

пунктов.

4. Повторите реализацию п.п. 1, 2 и 3 с помощью ресурсов, описывающих меню.

Работа с базами данных в Android

Механизм работы с базами данных в Android позволяет хранить и обрабатывать структурированную информацию. Любое приложение может создавать свои собственные базы данных, над которыми оно будет иметь полный контроль.
В Android используется библиотека *SQLite*, представляющую из себя реляционную СУБД, обладающую следующими характерными особенностями: свободно распространяемая (open source), поддерживающая стандартный язык запросов и транзакции, легковесная, одноуровневая (встраиваемая), отказоустойчивая.

Курсоры (Cursor) и ContentValues

Запросы к СУБД возвращают объекты типа *Cursor*. Для экономии ресурсов используется подход, когда при извлечении данных не возвращаются копии их значений из СУБД, а создается *Cursor*, обеспечивающий навигацию и доступ к запрошенному набору исходных данных. Методы объекта Cursor предоставляют различные возможности навигации, назначение которых, как правило, понятно из названия:

- moveToFirst
- moveToNext
- moveToPrevious
- getCount
- getColumnIndexOrThrow
- getColumnName
- getColumnNames
- moveToPosition
- *getPosition*

При добавлении данных в таблицы СУБД применяются объекты класса *ContentValues*. Каждый такой объект содержит данные одной строки в таблице и, по сути, является ассоциативным массивом с именами столбцов и соответствующими значениями.

Работа с СУБД SQLite

При создании приложений, использующих СУБД, во многих случаях удобно применять инструменты, называющиеся ORM (Object-Relationship Mapping), отображающие данные из одной или нескольких таблиц на объекты используемого языка программирования. Кроме того, ORM позволяют абстрагироваться от конкретной реализации и структуры таблиц и берут на себя обязанности по взаимодействию с СУБД. К сожалению, в силу ограниченности ресурсов мобильной платформы ORM в настоящий момент в Android практически не применяется. Тем не менее, разумным подходом при разработке приложения будет инкапсуляция всех взаимодействий с СУБД в одном классе,

методы которого будут предоставлять необходимые услуги остальным компонентам приложения.
Хорошей практикой является создание вспомогательного класса, берущего на себя работу с СУБД. Данный класс обычно инкапсулирует взаимодействия с базой данных, предоставляя интуитивно понятный строго типизированный способ удаления, добавления и изменения объектов. Такой *Адаптер базы данных* также должен обрабатывать запросы к БД и переопределять методы для открытия, закрытия и создания базы данных. Его также обычно используют как удобное место для хранения статических констант, относящихся к базе данных, таких, например, как имена таблиц и полей. Ниже показан пример каркаса для реализации подобного Адаптера:

```
public class SampleDBAdapter {
     private static final String DATABASE_NAME =
          "SampleDatabase.db";
     private static final String DATABASE_TABLE = "SampleTable";
     private static final int DATABASE_VERSION = 1;

     // Имя поля индекса для
     public static final String KEY_ID = "_id";

     // Название и номер п/п (индекс) каждого поля
     public static final String KEY_NAME = "name";
     public static final int NAME_COLUMN = 1;
     // Для каждого поля опишите константы аналогичным образом...

     // SQL-запрос для создания БД
     private static final String DATABASE_CREATE = "create table "
               + DATABASE_TABLE + " (" + KEY_ID
               + " integer primary key autoincrement, " + KEY_NAME
               + " text not null);";

     // Переменная для хранения объекта БД
     private SQLiteDatabase db;

     // Контекст приложения для
     private final Context context;

     // Экземпляр вспомогательного класса для
     // открытия и обновления БД
     private myDbHelper dbHelper;

     // Конструктор
     public SampleDBAdapter(Context _context) {
          context = _context;
          dbHelper = new myDbHelper(context, DATABASE_NAME, null,
                    DATABASE_VERSION);
     }
```

```
// «Открывашка» БД
public SampleDBAdapter open() throws SQLException {
     try {
          db = dbHelper.getWritableDatabase();
     } catch (SQLiteException e) {
          db = dbHelper.getReadableDatabase();
     }
     return this;
}

// Метод для закрытия БД
public void close() {
     db.close();
}

// Метод для добавления данных, возвращает индекс
// свежевставленного объекта
public long insertEntry(SampleObject _SampleObject) {
     // Здесь создается объект ContentValues, содержащий
     // нужные поля и  производится вставка
     return index;
}

// Метод для удаления строки таблицы по индексу
public boolean removeEntry(long _rowIndex) {
     return db.delete(DATABASE_TABLE, KEY_ID + "=" +
          _rowIndex, null) > 0;
}

// Метод для получения всех данных.
// Возвращает курсор, который можно использовать для
// привязки к адаптерам типа SimpleCursorAdapter
public Cursor getAllEntries() {
     return db.query(DATABASE_TABLE, new String[] { KEY_ID,
          KEY_NAME }, null, null, null, null, null);
}

// Возвращает экземпляр объекта по индексу
public SampleObject getEntry(long _rowIndex) {
     // Получите курсор, указывающий на нужные данные из БД
     // и создайте новый объект, используя этими данными
     // Если ничего не найдено, верните null
     return objectInstance;
}

// Изменяет объект по индексу
// Увы, это не ORM :(
public boolean updateEntry(long _rowIndex, SampleObject
     _SampleObject) {
     // Создайте объект ContentValues на основе
     // свойств SampleObject
```

```
            // и используйте его для обновления строки в таблице
            return true; // Если удалось обновить, иначе false :)
        }

        // Вспомогательный класс для открытия и обновления БД
        private static class myDbHelper extends SQLiteOpenHelper {
            public myDbHelper(Context context, String name,
                CursorFactory factory,   int version) {
                super(context, name, factory, version);
            }

            // Вызывается при необходимости создания БД
            @Override
            public void onCreate(SQLiteDatabase _db) {
                _db.execSQL(DATABASE_CREATE);
            }

            // Вызывается для обновления БД, когда текущая версия БД
            // в приложении новее, чем у БД на диске
            @Override
            public void onUpgrade(SQLiteDatabase _db,
                int _oldVersion, int _newVersion) {

                // Выдача сообщения в журнал, полезно при отладке
                Log.w("TaskDBAdapter", "Upgrading from version "
                    + _oldVersion
                    + " to " + _newVersion
                    + ", which will destroy all old data");

                // Обновляем БД до новой версии.
                // В простейшем случае убиваем старую БД
                // и заново создаем новую.
                // В реальной жизни стоит подумать о пользователях
                // вашего приложения и их реакцию на потерю
                // старых данных.
                _db.execSQL("DROP TABLE IF EXISTS "
                    + DATABASE_TABLE);
                onCreate(_db);
            }
        }
}
```

Работа с СУБД без класса-адаптера

При нежелании использовать класс *SQLiteOpenHelper* (и вообще адаптер БД) можно воспользоваться методом *openOrCreateDatabase* контекста приложения:

```
        private static final String DATABASE_NAME = "myDatabase.db";
        private static final String DATABASE_TABLE = "mainTable";
        private static final String DATABASE_CREATE = "create table "
```

```
        + DATABASE_TABLE
        + " ( _id integer primary key autoincrement,"
        + "column_one text not null);";
    SQLiteDatabase myDatabase;

    private void createDatabase() {
        myDatabase = openOrCreateDatabase(DATABASE_NAME,
            MODE_PRIVATE, null);
        myDatabase.execSQL(DATABASE_CREATE);
    }
```

В этом случае можно работать с базой данных с помощью, например, метода *execSQL* экземпляра БД, как показано на примере выше.

Особенности работы с БД в Android

При работе с базами данных в Android *следует избегать* хранения BLOB'ов с таблицах из-за резкого падения эффективности работы.
Как показано в примере адаптера БД, для каждой таблицы *рекомендуется* создавать автоикрементное поле **_id**, которое будет уникальным индексом для строк. Если же планируется делегировать доступ к БД с помощью контент-провайдеров, такое поле является *обязательным*.

Выполнение запросов для доступа к данным

Для повышения эффективности использования ресурсов мобильной платформы запросы к БД возвращают объект типа *Cursor*, используемый в дальнейшем для навигации и получения значений полей. Выполнение запросов осуществляется с помощью метода *query* экземпляра БД, параметры которого позволяют гибко управлять критериями выборки:

```
// Получить поля индекса, а также первое и третье
// поле из таблицы, без дубликатов
String[] result_columns = new String[] {KEY_ID, KEY_COL1,
    KEY_COL3};
Cursor allRows = myDatabase.query(true, DATABASE_TABLE,
    result_columns, null, null, null, null, null, null);

// Получить все поля для строк, где третье поле
// равно требуемому
// значению, результат отсортировать по пятому полю
String where = KEY_COL3 + "=" + requiredValue;
String order = KEY_COL5;
Cursor myResult = myDatabase.query(DATABASE_TABLE, null,
    where, null, null, null, order);
```

Доступ к результатам с помощью курсора

Для получения результатов запроса необходимо установить курсор на нужную строку с помощью методов вида moveToМестоположение, перечисленных выше. После этого используются типизированные методы getТип, получающие в качестве параметра индекс (номер) поля в строке. Как правило, эти значения являются статическими константами адаптера БД:

```
String columnValue = myResult.getString(columnIndex);
```

В примере ниже показано, как можно просуммировать все поля (типа float) из результатов выполнения запроса (и получить среднюю сумму счета):

```
int KEY_AMOUNT = 4;
Cursor myAccounts = myDatabase.query("my_bank_accounts", null,
          null, null, null, null, null);
float totalAmount = 0f;

// Убеждаемся, что курсор непустой
if (myAccounts.moveToFirst()) {
     // Проходимся по каждой строке
     do {
          float amount = myAccounts.getFloat(KEY_AMOUNT);
          totalAmount += amount;
     } while(myAccounts.moveToNext());
}

float averageAmount = totalAmount / myAccounts.getCount();
```

Изменение данных в БД

В классе *SQLiteDatabase*, содержащем методы для работы с БД, имеются методы *insert*, *update* и *delete*, которые инкапсулируют операторы SQL, необходимые для выполнения соответствующих действий. Кроме того, метод *execSQL* позволяет выполнить любой допустимый код SQL(если вы, например, захотите увеличить долю ручного труда при создании приложения). Следует помнить, что при наличии активных курсоров после любого изменения данных следует вызывать метод refreshQuery для всех курсоров, которые имеют отношение к изменяемым данным (таблицам).

Вставка строк

Метод insert хочет получать (кроме других параметров) объект ContentValues, содержащий значения полей вставляемой строки и возвращает значение индекса:

```
ContentValues newRow = new ContentValues();
```

```
// Выполним для каждого нужного поля в строке
newRow.put(COLUMN_NAME, columnValue);

db.insert(DATABASE_TABLE, null, newRow);
```

Обновление строк

Также используется *ContentValues*, содержащий подлежащие изменению поля, а для указания, какие именно строки нужно изменить, используется параметр *where*, имеющий стандартный для SQL вид:

```
ContentValues updatedValues = new ContentValues();

// Повторяем для всех нужных полей
updatedValues.put(COLUMN_NAME, newValue);

// Указываем условие
String where = COLUMN_NAME + "=" + "'Бармаглот'";

// Обновляем
db.update(DATABASE_TABLE, updatedValues, where, null);
```

Метод *update* возвращает количество измененных строк. Если в качестве параметра *where* передать *null*, будут изменены все строки таблицы.

Удаление строк

Выполняет похожим на *update* образом:

```
db.delete(DATABASE_TABLE, KEY_ID + "=" + rowId, null);
```

Использование SimpleCursorAdapter

SimpleCursorAdapter позволяет привязать *курсор* к *ListView*. используя описание разметки для отображения строк и полей. Для однозначного определения того, в каких элементах разметки какие поля, получаемые через курсор, следует отображать, используются два массива: строковый с именами полей строк, и целочисленный с идентификаторами элементов разметки:

```
Cursor cursor = [ . . . запрос к БД . . . ];

String[] fromColumns = new String[] {KEY_NAME, KEY_NUMBER};
int[] toLayoutIDs = new int[] { R.id.nameTextView,
     R.id.numberTextView};

SimpleCursorAdapter myAdapter;
myAdapter = new SimpleCursorAdapter(this,
     R.layout.item_layout, cursor, fromColumns, toLayoutIDs);
```

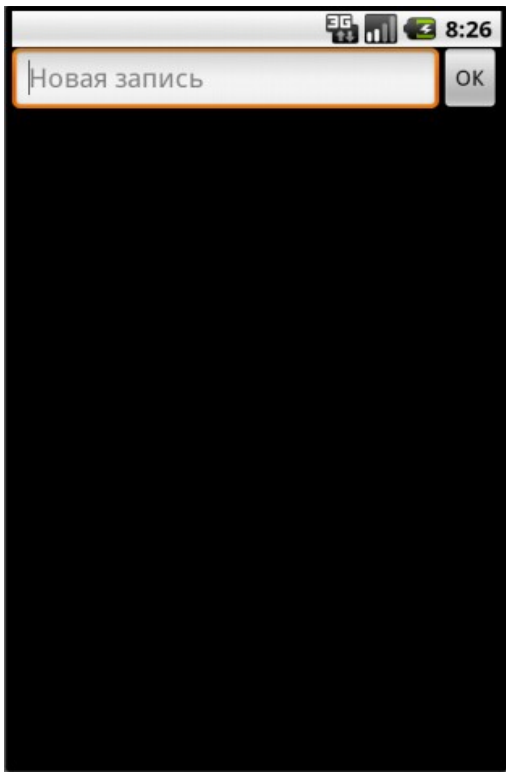
```
    myListView.setAdapter(myAdapter);
```

Лабораторная работа «работа с SQLite»

Целью работы является создание простого приложения, позволяющего создавать, хранить и удалять короткие заметки.

1. Создайте новый проект **NotesSample**.
2. Для простоты использования все действия с записями будут производиться в рамках одной Активности, поэтому используйте максимально упрощенный интерфейс, как показано на следующей странице. Для реализации можно воспользоваться такой разметкой в файле **res/layout/main.xml**:

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

    <RelativeLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" >

        <Button
            android:id="@+id/save_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentRight="true"
            android:text="@android:string/ok" />

        <EditText
            android:id="@+id/edit_text"
            android:layout_width="fill_parent"
            android:layout_toLeftOf="@id/save_button"
            android:layout_height="wrap_content"
            android:hint="Новая запись" />
    </RelativeLayout>

    <ListView
        android:id="@+id/myListView"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />

</LinearLayout>
```



3. Реализуйте сохранение записей по нажатию на кнопку, а удаление через контекстное меню. Работа с СУБД должна осуществляться с использованием адаптера. Пример обработчика выбора пункта контекстного меню покажет преподаватель.

Контент-провайдеры

Контент-провайдеры предоставляют *интерфейс для публикации и потребления* структурированных наборов данных, основанный на URI с использованием простой схемы *content://*. Их использование позволяет отделить код приложения от данных, делая программу менее чувствительной к изменениям в источниках данных.
Для взаимодействия с контент-провайдером используется уникальный URI, который обычно формируется следующим образом:

```
content://<домен-разработчика-наоборот>.provider.<имя-
приложения>/<путь-к-данным>
```

Классы, реализующие контент-провайдеры, чаще всего имеют статическую строковую константу CONTENT_URI, которая используется для обращения к данному контент-провайдеру.
Контент-провайдеры являются единственным способом доступа к данным других приложений и используются для получения результатов запросов, обновления, добавления и удаления данных. Если у приложения есть нужные полномочия, оно может запрашивать и модифицировать соответствующие данные, принадлежащие другому приложению, в том числе данные стандартных БД Android. В общем случае, контент-провайдеры следует создавать только тогда, когда требуется предоставить другим приложениям доступ к данным вашего приложения. В остальных случаях рекомендуется использовать СУБД (SQLite). Тем не менее, иногда контент-провайдеры используются внутри одного приложения для поиска и обработки специфических запросов к данным.

Использование контент-провайдеров

Для доступа к данным какого-либо контент-провайдера используется объект класса *ContentResolver,* который можно получить с помощью метода *getContentResolver* контекста приложения для связи с поставщиком в качестве клиента:

```
ContentResolver cr = getApplicationContext()
     .getContentResolver();
```

Объект *ContentResolver* взаимодействует с объектом контент-провайдера, отправляя ему запросы клиента. Контент-провайдер обрабатывает запросы и возвращает результаты обработки.
Контент-провайдеры представляют свои данные потребителям в виде одной или нескольких таблиц подобно таблицам реляционных БД. Каждая строка при этом является отдельным «объектом» со свойствами, указанными в соответствующих именованных полях. Как правило, каждая строка имеет уникальный

целочисленный индекс с именем «**_id**», который служит для однозначной идентификации требуемого объекта.
Контент-провайдеры, обычно предоставляют минимум два URI для работы с данными: один для запросов, требующих все данные сразу, а другой – для обращения к конкретной «строке». В последнем случае в конце URI добавляется /<номер-строки> (который совпадает с индексом «**_id**»).

Запросы на получение данных

Запросы на получение данных похожи на запросы к БД, при этом используется метод *query* объекта *ContentResolver*. Ответ также приходит в виде курсора, «нацеленного» на результирующий набор данных (выбранные строки таблицы):

```
ContentResolver cr = getContentResolver();

// получить данные всех контактов
Cursor c = cr.query(ContactsContract.Contacts.CONTENT_URI, null,
     null, null, null);

// получить все строки, где третье поле имеет конкретное
// значение и отсортировать по пятому полю
String where = KEY_COLUMN3 + "=" + requiredValue;
String order = KEY_COLUMN5;

Cursor someRows = cr.query(MyProvider.CONTENT_URI, null, where,
     null, order);
```

Изменение данных

Для изменения данных применяются методы *insert*, *update* и *delete* объекта ContentResolver. Для массовой вставки также существует метод *bulkInsert*.
Пример добавления данных:

```
ContentResolver cr = getContentResolver();

ContentValues newRow = new ContentValues();
// повторяем для каждого поля в строке:
newRow.put(COLUMN_NAME, newValue);
Uri myRowUri = cr.insert(SampleProvider.CONTENT_URI, newRow);

// Массовая вставка:
ContentValues[] valueArray = new ContentValues[5];

// здесь заполняем массив
// делаем вставку
int count = cr.bulkInsert(MyProvider.CONTENT_URI,
valueArray);
```

При вставке одного элемента метод *insert* возвращает URI вставленного элемента, а при массовой вставке возвращается количество вставленных элементов.
Пример удаления:

```
ContentResolver cr = getContentResolver();
// удаление конкретной строки
cr.delete(myRowUri, null, null);

// удаление нескольких строк
String where = "_id < 5";
cr.delete(MyProvider.CONTENT_URI, where, null);
```

Пример изменения:

```
ContentValues newValues = new ContentValues();

newValues.put(COLUMN_NAME, newValue);
String where = "_id < 5";

getContentResolver().update(MyProvider.CONTENT_URI, newValues,
     where, null);
```

Лабораторная работа «получение списка контактов»

Для чтения информации о контактах используется контент-провайдер *ContactsContract*, точнее, один из его подклассов. Для этой лабораторной работы воспользуемся провайдером **ContactsContract.Contacts**. Для чтения контактов приложению требуются полномочия **READ_CONTACTS**.
Важно! По умолчанию, все рассмотренные выше методы Активностей работают в главном потоке (*Main Thread*) и должны, в идеале, заканчиваться моментально, т. к. «замораживание» главного потока недопустимо. Для выполнения длительных операций должны использоваться вспомогательные/рабочие потоки (*Worker Threads*). С другой стороны, результатом запроса к Контент-провайдеру является объект типа *Cursor*, жизненный цикл которого обычно не совпадает с жизненным циклом Активности или Фрагмента, где этот курсор используется. Для «автоматизации» решения обеих задач (выполнения длительных действий и управления курсорами) в *Android* широко используются *Загрузчики* (*Loaders*), которые входят в программу следующего курса.

1. Добавьте несколько контактов в эмуляторе (поскольку требуется только отображаемое имя контакта, остальные поля можно не заполнять :).
2. Создайте новый проект **ContactsSample**.
3. Выведите имена всех контактов (с помощью ListView), используя для получения информации URI ContactsContract.Contacts.*CONTENT_URI*.
Необходимое имя поля для привязки адаптера найдите среди статических

констант класса ContactsContract.Contacts.

Создание контент-провайдеров (дополнительный материал)

Для создания собственного контент-провайдера требуется расширить класс *ContentProvider* и переопределить метод *onCreate*, чтобы проинициализировать источник данных, который требуется опубликовать. Остальные методы этого класса будут, по сути, обертками вокруг методов работы с исходным источником данных. Каркас для класса показан ниже:

```
public class NewProvider extends ContentProvider {

     public final static Uri CONTENT_URI=Uri.parse(
          "URI провайдера");

     @Override
     public int delete(Uri uri, String selection, String[]
     selectionArgs) {
          // Удаление данных
          return 0;
     }

     @Override
     public String getType(Uri uri) {
          // Возвращает тип MIME для указанных объектов(объекта)
          return null;
     }

     @Override
     public Uri insert(Uri uri, ContentValues values) {
          // Добавление данных
          return null;
     }

     @Override
     public boolean onCreate() {
          // Инициализация источника данных
          return false;
     }

     @Override
     public Cursor query(Uri uri, String[] projection, String
          selection, String[] selectionArgs, String sortOrder) {
          // Стандартный запрос
          return null;
     }

     @Override
     public int update(Uri uri, ContentValues values, String
          selection, String[] selectionArgs) {
          // Обновление данных
```

```
            return 0;
        }
    }
```

Как говорилось выше, URI обычно выглядят примерно так: *content://<домен-разработчика-наоборот>.provider.<имя-приложения>/<путь-к-даным>*
Традиционно URI должны быть представлены двумя способами, одним – для доступа ко всем значениям определенного типа, другой – для указания на конкретный экземпляр данных.
Например, URI *content://com.android.provider.countrypicker/stations* мог бы использоваться для получения списка всех станций (метро), а *content://com.android.provider.countrypicker/stations/17* – для станции с индексом 17.
При создании контент-провайдера обычно применяется статический объект класса *UriMatcher*, который служит для определения деталей запроса к контент-провайдеру. Использование *UriMatcher* позволяет «разгрузить» логику программы и избежать множественного сравнения строковых объектов за счет централизованного «отображения» *URI* разных видов на целочисленные константы. Особенно он полезен в случаях, когда создаваемый контент-провайдер обслуживает разные *URI* для доступа к одному и тому же источнику данных. Использовать *UriMatcher* очень просто: внутри класса контент-провайдера можно добавить такой код:

```
// Константы для разных типов запросов
private static final int ALL_STATIONS = 1;
private static final int SINGLE_STATION = 2;

private static final UriMatcher uriMatcher;

// Заполнение UriMatcher'a
// Если URI оканчивается на /stations - это запрос
// про все станции
// Если на stations/[ID] - про конкретную станцию
static {
     uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
     uriMatcher.addURI("com.example.provider.countrypicker",
          "stations", ALL_STATIONS);
     uriMatcher.addURI("com.example.provider.countrypicker",
          "stations/#", SINGLE_STATION);
}
```

В дальнейшем полученный в запросе к контент-провайдеру *URI* проверяется в методах класса следующим образом:

```
@Override
```

```
public Cursor query(Uri uri, String[] projection, String
selection, String[] selectionArgs, String sortOrder) {

     switch (uriMatcher.match(uri)) {
     case ALL_STATIONS:
          // Вернуть курсор, указывающий на выборку со
          // всеми станциями

     case SINGLE_STATION:
          // Вытащить ID станции из URI:
          String _id = uri.getPathSegments().get(1);

          // Вернуть курсор, указывающий на выборку с одной
          // станцией.

     }
     return null;
}
```

При наполнении объекта UriMatcher шаблонами могут применять в «#» и «*» в качестве специальных символов: # в шаблоне совпадает с любым *числом*, а * - с любым *текстом*.
Кроме уникальных URI, используемым создаваемым контент-провайдером, для определения клиентами типа возвращаемых по запросу данных используются уникальные типы MIME. Метод *getType* класса *ContentProvider* обычно возвращает один тип данных для массовой выборки, а другой – для одиночных записей. В нашем случае это могло бы выглядеть так:

```
@Override
public String getType(Uri uri) {
     switch (uriMatcher.match(uri)) {
     case ALL_STATIONS:
          return "vnd.com.example.cursor.dir/station";
     case SINGLE_STATION:
          return "vnd.com.example.cursor.item/station";
     default:
          throw new IllegalArgumentException("Unsupported URI: "
               + uri);
     }
}
```

Для того, что бы *Android* знал о существовании и мог использовать (через *ContentResolver*) ваш контент-провайдер, его нужно описать в Манифесте приложения. Минимальное описание выглядит так:

```
<provider android:name=".NewProvider"
     android:authorities="com.android.provider.countrypicker" >
 </provider>
```

В данном случае указаны только обязательные атрибуты элемента *<provider>:* имя класса контент-провайдера и его область ответственности. Более полную информацию о возможных атрибутах можно получить на сайте: http://d.android.com/guide/topics/manifest/provider-element.html

Использование интернет-сервисов

Помимо простого отображения web-контента с помощью виджета WebView, разработчик имеет возможность «низкоуровневой работы с разнообразными сетевыми сервисами. Для этого всего лишь требуется создать сетевое подключение (запрос к серверу), получить, обработать и отобразить данные к нужном виде. Традиционно наиболее удобными форматами для сетевого обмена данными считаются *XML* и *JSON*.
Разумеется, любое приложение, использующее сетевые подключения, должно иметь в Манифесте соответствующие полномочия:

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

Для создания потока данных от сервера можно использовать класс **HttpUrlConnection**, являющийся расширением класса *UrlConnection* из пакета *java.net*. Пакеты *java.net* и *android.net* содержат классы, позволяющие управлять сетевыми соединениями. Более подробную информацию о классе *HttpUrlConnection* с примерами использования можно получить здесь: **http://developer.android.com/reference/java/net/HttpURLConnection.html**
Простой пример создания соединения:

```
private static final String some_url = "....";
....
try {
     // Создаем объект типа URL
     URL url = new URL(getString(R.string.rates_url));

     // Соединяемся
     HttpURLConnection httpConnection = (HttpURLConnection)
          url.openConnection();

     // Получаем код ответа
     int responseCode = httpConnection.getResponseCode();

     // Если код ответа хороший, парсим поток(ответ сервера)
     if (responseCode == HttpURLConnection.HTTP_OK) {
          // Если код ответа хороший, обрабатываем ответ
          InputStream in = httpConnection.getInputStream();
     } else {
          // Сделать извещения об ошибках, если
          // код ответа нехороший
     }

} catch (MalformedURLException e) {
     e.printStackTrace();
} catch (IOException e) {
     e.printStackTrace();
```

```
}
```

Лабораторная работа «Использование сетевых сервисов»

Целью данной работы является создание простого приложения, получающего курсы иностранных валют по отношению к рублю с сайта ЦБ РФ в формате XML и отображающего данные в виде списка (*ListView*). Для получения данных будет использоваться URL **http://www.cbr.ru/scripts/XML_daily.asp** Ответ сервера выглядит примерно так:

```
<ValCurs Date="04.04.2012" name="Foreign Currency Market">
     <Valute ID="R01010">
          <NumCode>036</NumCode>
          <CharCode>AUD</CharCode>
          <Nominal>1</Nominal>
          <Name>Австралийский доллар</Name>
          <Value>30,4632</Value>
     </Valute>
     . . . . . .
</ValCurs>
```

Для каждой валюты (элемент *Value*) потребуется извлечь значения дочерних элементов *CharCode*, *Nominal*, *Name* и *Value*. Значение атрибута *Date* корневого элемента (*ValCurs*) будет использоваться для изменения заголовка приложения.

1. Создайте новый проект **CurrencyRates**. Главная (и единственная) Активность с таким же именем (CurrencyRates) должна расширять класс **ListActivity**.
2. Отредактируйте Манифест приложения, добавив в него необходимые полномочия и
   тему для Активности
   (`android:theme="@android:style/Theme.Light"`).
3. Файл строковых ресурсов **strings.xml** (в каталоге **res/values**) отредактируйте следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Курсы ЦБ РФ</string>
  <string
name="rates_url">http://www.cbr.ru/scripts/XML_daily.asp</string>
</resources>
```

4. Для отображения информации требуется создать разметку для элементов списка. В каталоге **res/layout** создайте файл **item_view.xml** со следующим содержимым:

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

    <TextView
        android:id="@+id/charCodeView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="#FF8"
        android:minWidth="45sp"
        android:padding="4dp"
        android:textColor="#00F"
        android:textStyle="bold"
        android:gravity="center"
        android:shadowDx="8"
        android:shadowDy="8"
        android:shadowColor="#000"
        android:shadowRadius="8"/>

    <TextView
        android:id="@+id/valueView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textColor="#008"
        android:background="#FFE"
        android:minEms="3"
        android:padding="3dp" />

    <TextView
        android:id="@+id/nominalView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="3dp" />

    <TextView
        android:id="@+id/nameView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:ellipsize="marquee"
        android:singleLine="true" />

</LinearLayout>
```

5. Вся логика приложения будет сосредоточена в классе *CurrencyRates*, поэтому остальные изменения будут касаться только этого класса. Добавьте необходимые константы:

```
    private final static String KEY_CHAR_CODE = "CharCode";
    private final static String KEY_VALUE = "Value";
    private final static String KEY_NOMINAL = "Nominal";
    private final static String KEY_NAME = "Name";
```

6. Тело метода onCreate должно содержать только две строки:

```
super.onCreate(savedInstanceState);
populate();
```

Поскольку *CurrencyRates* является наследником *ListActivity*, вызов *setContentView* не требуется. Метод *populate* будет наполнять *ListView* содержимым с помощью адаптера (*SimpleAdapter*), заполнив его данными, полученными от метода *getData*.

7. Добавьте метод **populate**, в котором создается и настраивается адаптер:

```
private void populate() {
    ArrayList<Map<String, String>> data = getData();

    String[] from = { KEY_CHAR_CODE, KEY_VALUE, KEY_NOMINAL,
                      KEY_NAME };
    int[] to = { R.id.charCodeView, R.id.valueView,
        R.id.nominalView, R.id.nameView };

    SimpleAdapter sa = new SimpleAdapter(this, data,
        R.layout.item_view, from, to);

    setListAdapter(sa);
}
```

8. Добавьте метод **getData**. Именно в нем будет создаваться и обрабатываться соединение с сервером, а также анализироваться XML-данные и заполняться список, который будет отображаться адаптером. Метод **getData** объемнее остальных, но ничего сложного в нем нет (стоит отметить, что интерфейсы *Document*, *Element* и *NodeList* должны импортироваться из пакета *org.w3c.dom*):

```
private ArrayList<Map<String, String>> getData() {
    ArrayList<Map<String, String>> list =
        new ArrayList<Map<String, String>>();
    Map<String, String> m;

    try {
        // Создаем объект URL
        URL url = new URL(getString(R.string.rates_url));

        // Соединяемся
        HttpURLConnection httpConnection =
            (HttpURLConnection) url.openConnection();
```

```
// Получаем от сервера код ответа
int responseCode = httpConnection.getResponseCode();

// Если код ответа хороший, парсим поток(ответ сервера),
// устанавливаем дату в заголовке приложения и
// заполняем list нужными Map'ами
if (responseCode == HttpURLConnection.HTTP_OK) {
     InputStream in = httpConnection.getInputStream();
     DocumentBuilderFactory dbf = DocumentBuilderFactory
               .newInstance();
     DocumentBuilder db = dbf.newDocumentBuilder();

     Document dom = db.parse(in);
     Element docElement = dom.getDocumentElement();
     String date = docElement.getAttribute("Date");
     setTitle(getTitle() + " на " + date);

     NodeList nodeList = docElement
          .getElementsByTagName("Valute");

     int count = nodeList.getLength();
     if (nodeList != null && count > 0) {
          for (int i = 0; i < count; i++) {
               Element entry = (Element) nodeList
                    .item(i);
               m = new HashMap<String, String>();

               String charCode = entry
                    .getElementsByTagName(KEY_CHAR_CODE)
                    .item(0).getFirstChild()
                    .getNodeValue();

               String value = entry
                    .getElementsByTagName(KEY_VALUE)
                    .item(0).getFirstChild()
                    .getNodeValue();

               String nominal = "за " + entry
                    .getElementsByTagName(KEY_NOMINAL)
                    .item(0).getFirstChild()
                    .getNodeValue();

               String name = entry
                    .getElementsByTagName(KEY_NAME)
                    .item(0).getFirstChild()
                    .getNodeValue();

               m.put(KEY_CHAR_CODE, charCode);
               m.put(KEY_VALUE, value);
               m.put(KEY_NOMINAL, nominal);
```

```
                        m.put(KEY_NAME, name);

                        list.add(m);
                    }
                }
            } else {
                // Сделать извещения об ошибках, если код ответа
                // нехороший
            }

        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParserConfigurationException e) {
            e.printStackTrace();
        } catch (SAXException e) {
            e.printStackTrace();
        }

        return list;
};
```

9. Проверьте работоспособность приложения. В реальной программе следует отслеживать наличие подключения устройства к сети, отслеживать ошибки соединения, а также получать данные из сети в *отдельном потоке*, чтобы не «замораживать» интерфейс пользователя. Эти темы будут рассмотрены во втором курсе.

Диалоги

Диалоги в Android являются небольшими модальными окнами, предлагающими пользователю принять то или иное решение перед тем, как работа приложения будет продолжена. Рекомендации по дизайну Диалогов можно найти по адресу http://developer.android.com/design/building-blocks/dialogs.html
Базовым классом для всех диалогов является *Dialog*, но его прямое использование не рекомендуется – вместо этого предлагается пользоваться классами *AlertDialog*, *DatePickerDialog* и *TimePickerDialog*. Кроме того, рекомендуется использовать класс *DialogFragment* в качестве контейнера для Диалогов, этот класс имеет всё необходимое для управления Диалогами и берет на себя заботу об обработке жизненного цикла Диалогов. Диалоги классов **PickerDialog* и их использование описаны официальном сайте: http://developer.android.com/guide/topics/ui/controls/pickers.html, а здесь далее будут рассмотрены приемы работы с объектами типа *AlertDialog*.

Создание Диалога

Для работы с Диалогом с использованием *DialogFragment* требуется расширить класс Фрагмента и переопределить его метод *onCreateDialog()*:

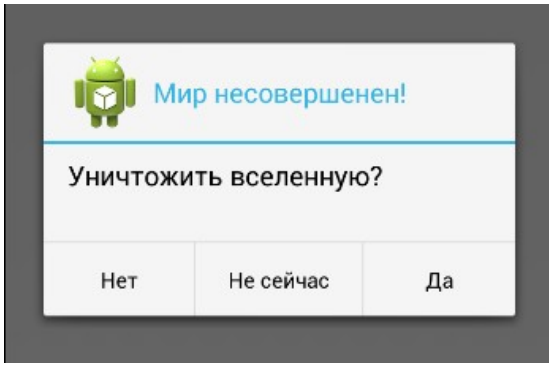
```
public class SampleDialogFragment extends DialogFragment {

     public SampleDialogFragment() {
          super();
     }

     @Override
     public Dialog onCreateDialog(Bundle savedInstanceState) {
          AlertDialog.Builder builder = new
               AlertDialog.Builder(getActivity());

          builder.setTitle("Мир несовершенен!")
               .setMessage("Уничтожить вселенную?")
               .setIcon(R.drawable.android_logo)
               .setPositiveButton("Да", new DialogInterface
                    .OnClickListener() {

                    @Override
                    public void onClick(DialogInterface dialog, int
                         which) {
                              // Уничтожаем вообще всё
                    }
               }).setNegativeButton("Нет",
               new DialogInterface.OnClickListener() {
```

```
                    @Override
                    public void onClick(DialogInterface dialog, int
                         which) {
                              // Тихо уходим
                    }
               }).setNeutralButton("Не сейчас",
               new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int
                         which) {
                              // Подождем
                    }
               });
          return builder.create();
     }
}
```

Для вывода такого Диалога на экран в Активности требуется вызвать метод show() нужного объекта:

```
SampleDialogFragment df = new SampleDialogFragment();
df.show(getSupportFragmentManager(), "SampleDialog");
```

В результате на экране должен появиться следующий Диалог:



Обратите внимание на то, что оформление Диалога, созданного показанным Выше способом, зависит от используемой визуальной Темы.
Стандартный отображаемый Диалог можно разделить на три области:

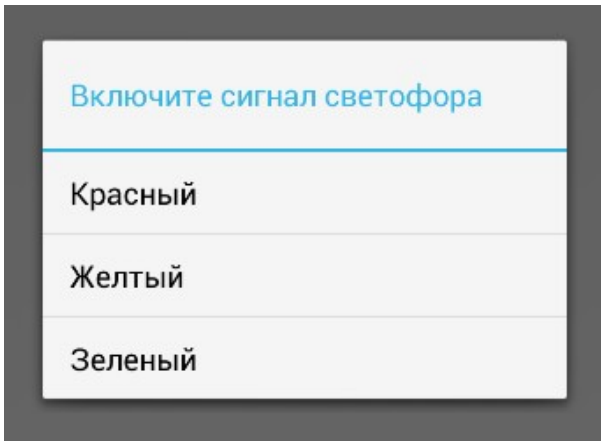
- Заголовок: содержимое устанавливается с помощью методов *setTitle()* и *setIcon()*;
- Область данных: может отображать сообщения (как на примере), списки выбора (включая радиогруппы и чекбоксы), а также произвольную разметку;
- Область кнопок: может отсутствовать вообще, а состав кнопок определяется использованными методами *set{Positive|Negative|Neutral}Button()*.

Использование списков выбора

Для добавления обычного списка выбора в Диалог используется метод *setItems()*:

```
final static CharSequence[] COLORS = { "Красный", "Желтый",
     "Зеленый", };

@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
     AlertDialog.Builder builder =
          new AlertDialog.Builder(getActivity());
     builder.setTitle("Включите сигнал светофора")
          .setItems(COLORS,
          new DialogInterface.OnClickListener() {
               @Override
               public void onClick(DialogInterface dialog,
                    int idx) {
                    // idx содержит индекс выбранного элемента
                    // списка
               }
     });

     return builder.create();
}
```

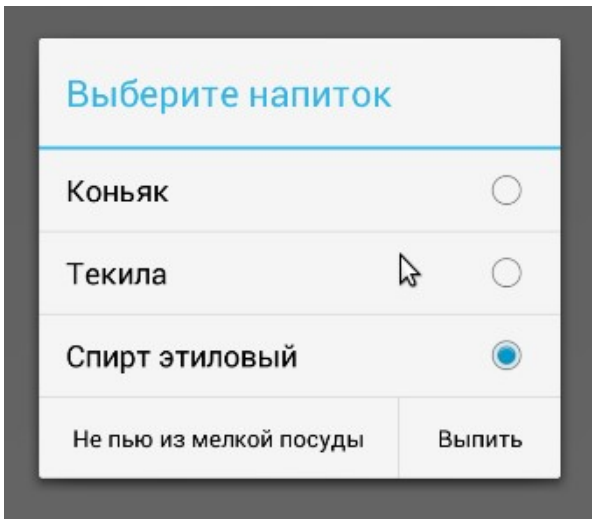
Как обычно, используемые списки рекомендуется хранить в ресурсах, на примере выше используется массив *CharSequence* исключительно для наглядности. При необходимости отображать в списке выбора элементы из динамического источника (например, БД) используется метод *setAdapter()*. В результате получается Диалог следующего вида:



Для добавления радиокнопок или чекбоксов в Диалог применяются методы *setSingleChoiceItems()* и *setMultiChoiceItems()*.
В методе *setSingleChoiceItems()* второй параметр – индекс «активной» радиокнопки:

```
final static CharSequence[] BEVERAGES = { "Коньяк", "Текила",
               "Спирт этиловый", };

@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
     AlertDialog.Builder builder = new AlertDialog
          .Builder(getActivity());
     builder.setTitle("Выберите напиток")
          .setSingleChoiceItems(BEVERAGES, -1,
               new DialogInterface.OnClickListener() {

                    @Override
                    public void onClick(DialogInterface dialog, int
                         idx) {
                    // Отмечаем выбранное
               }
          }).setPositiveButton("Выпить",
               new DialogInterface.OnClickListener(){

                    @Override
                    public void onClick(DialogInterface dialog, int
                         which) {
                    // Пьем
               }
          }).setNegativeButton("Не пью из мелкой посуды",
               new DialogInterface.OnClickListener() {

               @Override
               public void onClick(DialogInterface dialog, int
                         which) {
                    // Выливаем в пропасть
               }
          });

     return builder.create();
}
```

приводит к следующему результату:

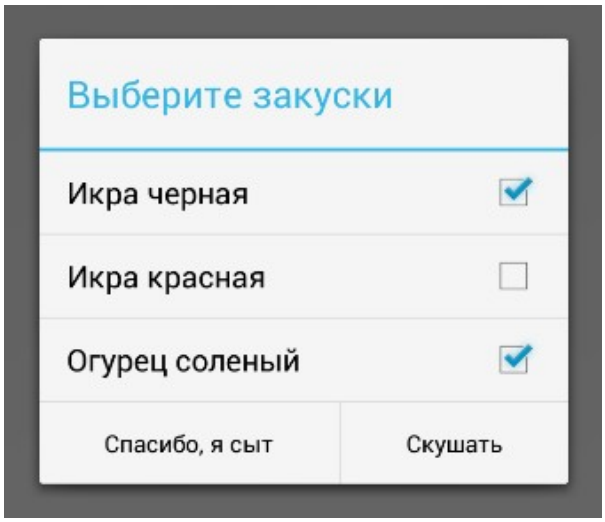


Аналогичным образом делается Диалог с чекбоксами:

```
final static CharSequence[] SNACKS = { "Икра черная", "Икра
красная",
                "Огурец соленый", };

@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
     AlertDialog.Builder builder = new
AlertDialog.Builder(getActivity());
          builder.setTitle("Выберите
закуски").setMultiChoiceItems(SNACKS,
                    null, new
DialogInterface.OnMultiChoiceClickListener() {

                @Override
                public void onClick(DialogInterface dialog, int
                          idx,
                     boolean isChecked) {
                          // Отмечаем выбранное
                     }
                }).setPositiveButton("Скушать",
                     new DialogInterface.OnClickListener() {

                     @Override
                     public void onClick(DialogInterface dialog, int
                          which) {
                          // Кушаем
                     }
                }).setNegativeButton("Спасибо, я сыт",
                     new DialogInterface.OnClickListener() {

                     @Override
                     public void onClick(DialogInterface dialog, int
                          which) {
```

```
                    // Выбрасываем закуски
                }
            });

    return builder.create();
}
```

Получаем такой вид:



Диалоги с произвольной разметкой

При необходимости в Диалоге можно использовать собственную разметку. Если дерево объектов UI создается динамически в коде, его можно использовать сразу, в противном случае (*XML*-ресурс) потребуется *LayoutInflater*. Метод *setView()* билдера позволяет назначить разметку Диалогу:

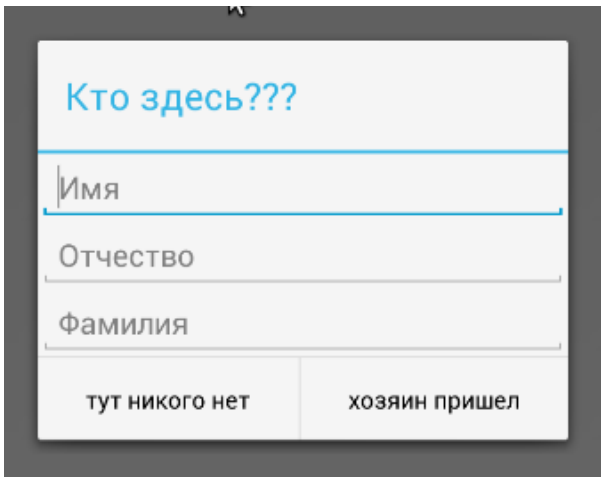
```
@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
    LayoutInflater inflater = getActivity().getLayoutInflater();
    AlertDialog.Builder builder = new AlertDialog
        .Builder(getActivity());
    builder.setTitle("Кто здесь???")
        .setView(inflater.inflate(R.layout.dialog_layout, null))
        .setPositiveButton("хозяин пришел",
            new DialogInterface.OnClickListener() {

            @Override
            public void onClick(DialogInterface dialog, int
                which) {
                // Сохраняем данные
            }
```

```
        }).setNegativeButton("тут никого нет",
             new DialogInterface.OnClickListener() {

             @Override
             public void onClick(DialogInterface dialog, int
                  which) {
                  // Тихо уходим
             }
        });

     return builder.create();
}
```

Файл разметки имеет обычное содержимое:

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

    <EditText
        android:id="@+id/forename"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Имя" />

    <EditText
        android:id="@+id/patronymic"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Отчество" />

    <EditText
        android:id="@+id/surename"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Фамилия" />

</LinearLayout>
```

В результате получаем Диалог нужного вида:



Обработка событий Диалога

При исчезновении Диалога с экрана вызывается его метод *onDismiss()*. Если пользователь отказался взаимодействовать с Диалогом (была нажата кнопка *Back* или пользователь коснулся экрана за пределами Диалога), перед *onDismiss()* также вызывается метод *onCancel()*. Убрать Диалог с экрана с последующими вызовами соответствующих обработчиков событий разработчик может, вызвав *dismiss()* или *cancel()*.

Лабораторная работа «Использование Диалога»

1. Создайте проект *DialogSample*.
2. В главной Активности реализуйте Диалог со списком из трех поисковых систем. Выбор элемента списка должен приводить к показу соответствующей web-страницы.
3. Реализуйте то же самое с помощью радиокнопок.
4. Добавьте в Диалог строку поиска и добейтесь, чтобы пользователю показывались результаты поиска в выбранной поисковой системе.

Широковещательные Приемники (Broadcast Receivers)

Широковещательные Приемники (далее – просто Приемники) в Android являются важным компонентом механизма передачи сообщений. Переносчиком сообщений являются объекты класса *Intent*, а сами сообщения посылаются с помощью методов *send*Broadcast()* класса *Context*. Для приема сообщений Приемник должен быть зарегистрирован либо в Манифесте приложения, либо в коде с помощью метода *registerReceiver()* класса Context. Зарегистрированный приемник будет принимать сообщения даже в случае, если приложение, частью которого он является, в настоящий момент не выполняется (Android запустит процесс с приложением, чтобы выполнить метод *onReceive()* Приемника). Динамически зарегистрированный Приемник остается зарегистрированным до явной его дерегистрации с помощью метода *unregisterReceiver()*. Какие сообщения каким Приемникам будут доставляться, Android определяет в результате *Intent Resolution* точно так же, как и в случае запуска Активностей и Сервисов. Это означает, что регистрация Приемника требует указания релевантных Фильтров Намерений (*Intent Filters*).

Без принятия специальных мер Приемник может принимать сообщения (и производить реализованные в нем действия) от любых приложений в системе. Для приемников, зарегистрированных в Манифесте, оградить Приемник от приема чужеродных сообщений можно с помощью нескольких действий:

- Использование собственного «пространства имен» в виде префиксов при создании имен Действий (*Action*) для Намерений и Фильтров Намерений снижает риск случайной коллизии сообщений;
- Добавление атрибута *android:permission* позволяет ограничить круг отправителей сообщений Приемнику только приложениями, у которых есть соответствующие полномочия (указаны в теге *uses-permission* в Манифесте);
- Установив атрибут *android:exported* в значение «*false*» для Приемника можно сделать его недоступным для прочих приложений.

Если Приемник должен принимать сообщения только внутри одного приложения и при этом регистрируется динамически, эффективной и безопасной заменой классу *Context* будет класс *LocalBroadcastManager* со своими методами [де]регистрации Приемников и передачи сообщений.

Создание Broadcast Receiver

Как правило, единственное что требуется для реализации Приемника – расширить класс *BroadcastReceiver* и переопределить метод *onReceive()*:

```
public class SampleReceiver extends BroadcastReceiver {
```

```
    @Override
    public void onReceive(Context context, Intent intent) {
        // Делаем что-нибудь полезное, но очень быстро
    }
}
```

Регистрация Приемника в Манифесте в самом простом случае выглядит так:

```
<receiver android:name=".SampleReceiver" >
    <intent-filter>
        <action  android:name=
     "com.example.receiversample.intent.action.SAMPLE_ACTION" />
    </intent-filter>
</receiver>
```

При регистрации в коде следует принимать во внимание жизненный цикл Активностей и других компонентов приложения, чтобы вовремя приемник разрегистрировать:

```
private static final String ACTION_SAMPLE_ACTION =
    "com.example.receiversample.intent.action.SAMPLE_ACTION";
private SampleReceiver mReceiver;

@Override
protected void onStart() {
    super.onStart();
    IntentFilter filter = new IntentFilter(ACTION_SAMPLE_ACTION);
    mReceiver = new SampleReceiver();
    registerReceiver(mReceiver, filter);
}

@Override
protected void onStop() {
    super.onStop();
    unregisterReceiver(mReceiver);
}
```

Послать сообщение позволяет метод sendBroadcast():

```
    sendBroadcast(new Intent(ACTION_SAMPLE_ACTION));
```

Лабораторная работа «Использование Широковещательных Приемников»

1. Создайте проект *ReceiverSample* и реализуйте в нем Приемник, показывающий при приеме сообщения с помощью *Toast*'а переданную с Намерении информацию.

2. Модифицируйте результат лабораторной работы «Использование Диалога» так, чтобы результаты работы Диалога передавались Приемнику.

Приложение A. Управляющие клавиши эмулятора

Emulated Device Key	Keyboard Key
Home	HOME
Menu (left softkey)	F2 *or* Page-up button
Star (right softkey)	Shift-F2 *or* Page Down
Back	ESC
Call/dial button	F3
Hangup/end call button	F4
Search	F5
Power button	F7
Audio volume up button	KEYPAD_PLUS, Ctrl-F5
Audio volume down button	KEYPAD_MINUS, Ctrl-F6
Camera button	Ctrl-KEYPAD_5, Ctrl-F3
Switch to previous layout orientation (for example, portrait, landscape)	KEYPAD_7, Ctrl-F11
Switch to next layout orientation (for example, portrait, landscape)	KEYPAD_9, Ctrl-F12
Toggle cell networking on/off	F8
Toggle code profiling	F9 (only with **-trace** startup option)
Toggle fullscreen mode	Alt-Enter
Toggle trackball mode	F6
Enter trackball mode temporarily (while key is pressed)	Delete
DPad left/up/right/down	KEYPAD_4/8/6/2
DPad center click	KEYPAD_5
Onion alpha increase/decrease	KEYPAD_MULTIPLY(*) / KEYPAD_DIVIDE(/)

Источник: **http://developer.android.com/tools/help/emulator.html**