

Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерного проектирования

Кафедра инженерной психологии и эргономики

ЭРГОНОМИКА МОБИЛЬНЫХ ПРИЛОЖЕНИЙ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

*Рекомендовано УМО по образованию
в области информатики и радиоэлектроники
в качестве пособия для специальности
11-58 01 01 «Инженерно-психологическое обеспечение
информационных технологий»*



Минск БГУИР 2017

УДК 331.101.1:004.42(076.5)
ББК (30.17+32.973.26-018.2)я73
Э74

Авторы:

М. М. Меженная, А. А. Быков, А. И. Каландаров, Т. В. Гордейчук

кафедра технологий программирования
Белорусского государственного университета
(протокол №10 от 16.03.2017);

доцент кафедры программного обеспечения вычислительной техники и
автоматизированных систем

Белорусского национального технического университета
кандидат технических наук, доцент Н. А. Разоренов

Эргономика мобильных приложений. Лабораторный практикум:
Э74 пособие / М. М. Меженная [и др.] – Минск : БГУИР, 2017. – 80 с. : ил.
ISBN 978-985-543-355-3.

Пособие посвящено вопросам проектирования интерфейсов и программирования мобильных приложений на платформе Android. Является методическим обеспечением выполнения лабораторных работ для студентов.

УДК 331.101.1:004.42(076.5)
ББК (30.17+32.973.26-018.2)я73

ISBN 978-985-543-355-3

© УО «Белорусский государственный
университет информатики
и радиоэлектроники», 2017

Содержание

Введение.....	4
Лабораторная работа №1	
Activity: работа с элементами экрана, обработка нажатий кнопок	5
Лабораторная работа №2	
Разработка приложений с несколькими Activity. Передача данных между Activity.....	22
Лабораторная работа №3	
Списки. Создание собственного адаптера. Механизмы обратного вызова.	39
Лабораторная работа №4	
Фрагменты. ViewPager. Хранение информации в базе данных SQLite	56

Введение

Настоящее пособие предназначено для проведения лабораторных работ для студентов специальности 1-58 01 01 «Инженерно-психологическое обеспечение информационных технологий», а также для студентов других специальностей, направленных на подготовку инженеров-программистов со знаниями основ разработки мобильных приложений.

Пособие посвящено проектированию интерфейсов и программированию мобильных приложений на платформе Android: описываются архитектура и основные компоненты системы Android (деятельность, служба, приемник широковещательных намерений, контент-провайдер); элементы экрана Android-приложения и их свойства, обработка событий в Activity; принципы работы с несколькими Activity в Android-приложениях; работа со всплывающими сообщениями, списками и фрагментами; механизмы обратного вызова для отслеживания событий в многофункциональных Android-приложениях. Подробно раскрываются вопросы хранения информации с использованием базы данных SQLite.

Лабораторная работа №1

Activity: работа с элементами экрана, обработка нажатий кнопок

Цель: формирование у студентов знаний и навыков работы с элементами экрана, обработки нажатий кнопок.

План занятия:

1. Изучить теоретические сведения.
2. Выполнить практическое задание по лабораторной работе.
3. Оформить отчет и ответить на контрольные вопросы.

Теоретические сведения

Мобильное приложение и мобильная платформа

Мобильное приложение представляет собой специально разработанное под конкретную мобильную платформу (Android, iOS, Windows Phone) программное обеспечение. Такое приложение также называют нативным мобильным приложением: его разрабатывают на языке высокого уровня и компилируют в нативный код операционной системы, дающий максимальную производительность и функциональность. Нативные мобильные приложения распространяются через магазины (AppStore, Windows Store, Google Play), существенно расширяют способы монетизации бизнеса по сравнению с веб-приложениями, однако характеризуются высокой стоимостью и длительным временем разработки.

Современные мобильные приложения создаются для линейки мобильных устройств: смартфонов, планшетов, электронных книг, цифровых проигрывателей, наручных часов, игровых приставок, нетбуков, смартбуков, очков, а также для телевизоров (тем самым выходя за рамки исключительно мобильных устройств). Работу таких устройств обеспечивает мобильная операционная система (мобильная платформа), сочетающая в себе функциональность операционной системы для персонального компьютера с функциями для мобильных и карманных устройств: сенсорный экран, сотовая связь, Bluetooth, Wi-Fi, GPS-навигация, камера, видеокамера, распознавание речи, диктофон, музыкальный плеер, NFC и инфракрасное дистанционное управление.

Наиболее распространенные операционные системы (платформы) для мобильных устройств: Android (платформа с открытым исходным кодом на основе ядра Linux и собственной реализации виртуальной машины Java от Google), iOS

(операционная система компании Apple, основанная на микроядре Mach и используемая в смартфонах iPhone), Windows Phone (разработка компании Microsoft).

Архитектура и основные компоненты мобильной платформы Android

В рамках дисциплины «Эргономика мобильных приложений» рассматриваются вопросы проектирования интерфейсов и разработки мобильных приложений под платформу Android, получившую наибольшее распространение в мире.

Архитектуру Android принято делить на уровни ядра, библиотек и среды выполнения, каркаса приложений, собственно приложений (рисунок 1.1).

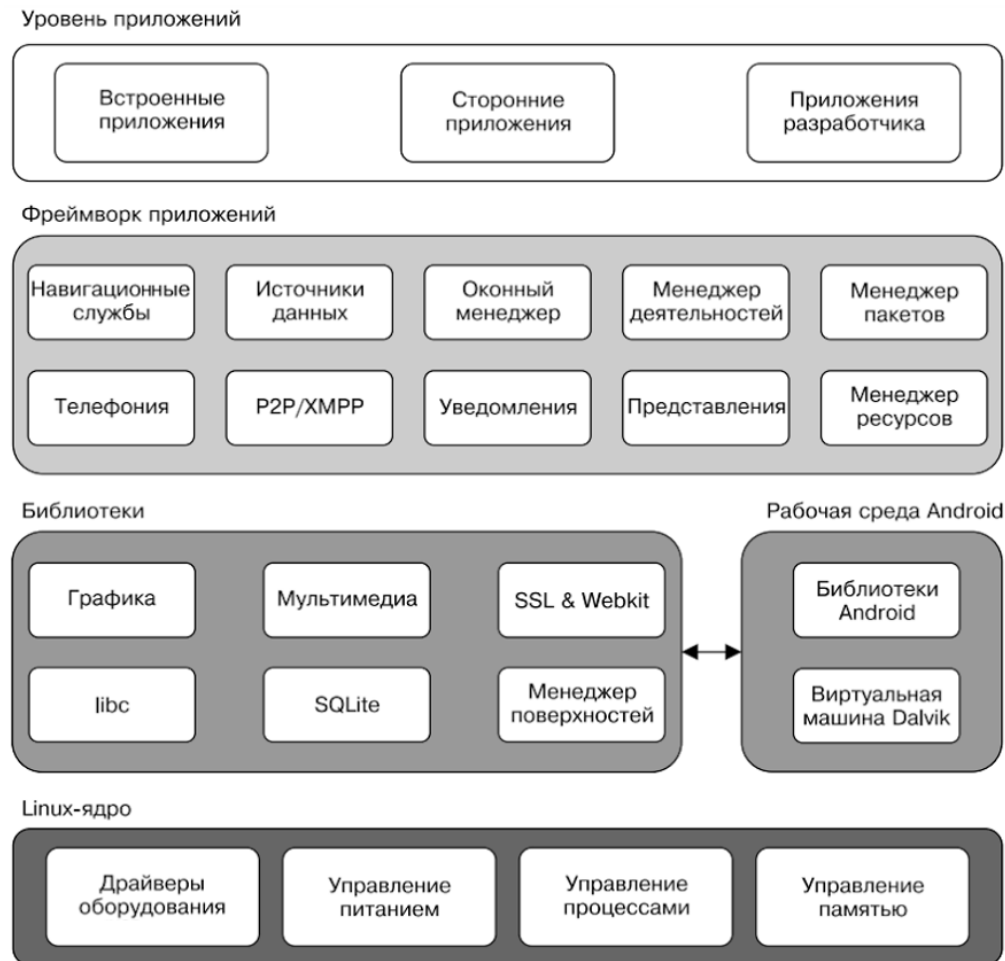


Рисунок 1.1 – Архитектура мобильной платформы Android

Ядро Linux обеспечивает функционирование системы и отвечает за безопасность, управление памятью, энергосистемой и процессами, а также предоставляет сетевой стек и модель драйверов.

Набор библиотек (Libraries) предназначен для обеспечения важнейшего базового функционала приложений и отвечает за поддержку файловых форматов,

кодирование и декодирование информации (например, цифровой звук и видео), отображение графики, поддержку веб-компонентов (WebKit), SQL-СУБД (SQLite) и стандартной для Linux-систем функциональности библиотек C. На этом же уровне располагается рабочая среда Android (Android Runtime). Каждое приложение в операционной системе Android запускается в собственном экземпляре виртуальной машины Dalvik. Таким образом, все работающие процессы изолированы от операционной системы и друг от друга. Благодаря этому осуществляется защита ядра операционной системы от возможного вреда со стороны других ее составляющих.

Уровень каркаса приложений (Application Framework) включает основные службы Android для управления жизненным циклом приложений, пакетами, ресурсами и т. д. Программист имеет полный доступ к тем же API (Application Programming Interface), которые используются основными приложениями. Архитектура этих приложений разработана с целью упрощения многократного использования компонентов. Любое разрабатываемое приложение может использовать возможности базовых приложений и, соответственно, любое другое стороннее приложение может использовать возможности вашего приложения (с учетом установленных разрешений).

Уровень приложений (Applications) включает стандартные приложения Android (браузер WebKit, календарь Google, клиент Gmail, приложение Gmaps, SMS-мессенджер и e-mail клиент), а также дополнительно загруженные приложения (из магазина Android). Android не делает разницы между основными приложениями и сторонним программным обеспечением, в связи с чем ключевые компоненты, такие как набор номера, рабочий стол или почтовый клиент Gmail, можно заменить альтернативными аналогами.

Всего в Android-приложениях существует четыре типа компонентов из уровня каркаса приложений: деятельность (Activity), служба (Service), приемник широковещательных намерений (Broadcast Receiver), контент-провайдер (Content Provider). Взаимодействие компонентов осуществляется с помощью объектов Intent.

Activity представляет собой визуальный пользовательский интерфейс для приложения – окно. Activity может также использовать дополнительные окна, например всплывающее диалоговое окно. Все деятельности реализуются как подкласс базового класса Activity.

Компонент Service не имеет визуального интерфейса пользователя и выполняется в фоновом режиме в течение неопределенного периода времени, пока не завершит свою работу. Service, как правило, требуется для длительных операций или для обеспечения работы удаленных процессов, но в общем случае это просто

режим, который функционирует, когда приложение не в фокусе. Примером такого процесса может стать прослушивание музыки в то время, когда пользователь делает что-то другое, или получение данных по сети без блокирования текущей активности. Приложения могут подключаться к компоненту Service или запускать его, если он не запущен, а также останавливать уже запущенные компоненты.

Broadcast Receiver используется для отслеживания внешних событий и реакции на них. Приложение может иметь несколько компонентов Broadcast Receiver, чтобы ответить на любые объявления, которые оно считает важными. При этом каждый компонент Broadcast Receiver будет определять код, который выполнится после возникновения конкретного внешнего события. С помощью класса Notification Manager можно сообщить пользователю информацию, требующую его внимания. Примером оповещений может быть сигнал о том, что информация загружена на устройство и доступна к использованию.

Content Provider делает определенный набор данных, используемых приложением, доступным для других приложений. Данные могут быть сохранены в файловой системе, базе данных SQLite, сети или любом другом месте, к которому приложение может иметь доступ. Контент-провайдеры для безопасного доступа к данным используют механизм разрешений. Посредством Content Provider другое приложение может запрашивать данные и, если выставлены соответствующие разрешения, изменять их. Например, система Android содержит Content Provider, который управляет пользовательской информацией о контактах.

Intent – специальные классы в коде приложения, которые определяют и описывают запросы приложения на выполнение каких-либо операций. Намерения добавляют слой, позволяющий оперировать компонентами с целью их повторного использования и замещения. В некоторых случаях это может быть очень мощным средством интеграции приложений. Главная особенность платформы Android состоит в том, что одно приложение может использовать элементы других приложений при условии, что эти приложения разрешают использовать свои компоненты. При этом ваше приложение не включает код другого приложения или ссылки на него, а просто запускает нужный элемент другого приложения [1-5].

Создание проекта в Android Studio

Разработку приложений для платформы Android выполняют на языке Java. С 2013 года разработка выполняется в официальной среде компании Google – Android Studio, основанной на IntelliJ IDEA от JetBrains.

Для создания нового проекта в Android Studio выбираем в окне приветствия опцию New Project (для уже открытого проекта в меню File следует выбрать New Project). В появившемся окне Create New Project (рисунок 1.2) заполняем поля имени приложения (Application name), которое будет отображаться для пользователей, и квалификатора (Company Domain), который будет добавляться к имени

пакета. Полное название проекта (Package name) формируется в соответствии с правилами именования пакетов в Java. При необходимости изменяем папку размещения проекта (Project location).

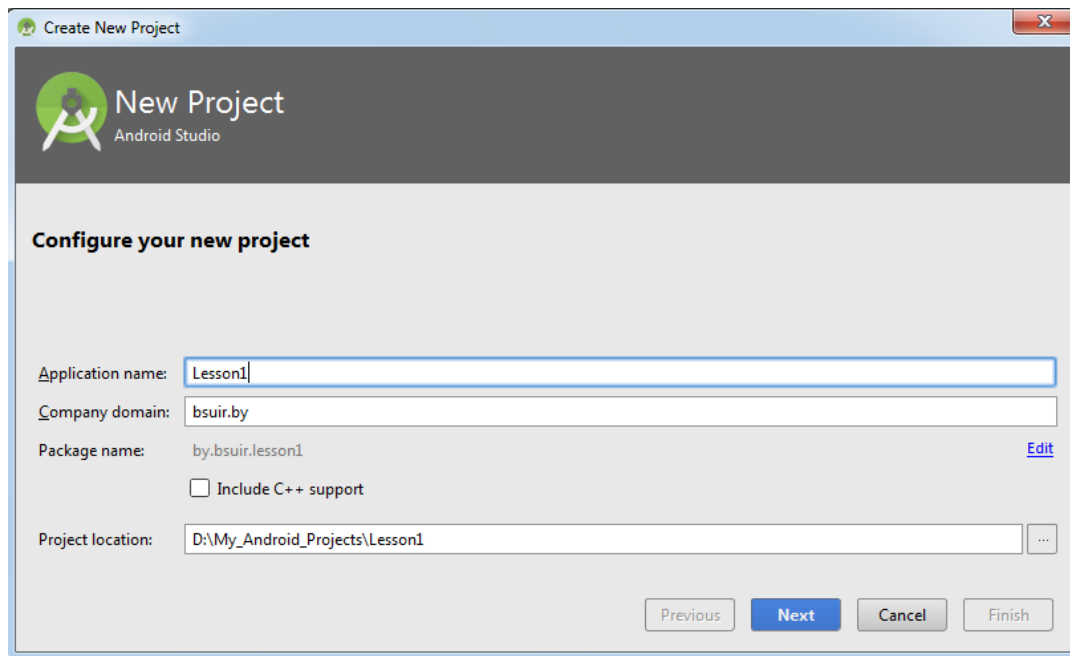


Рисунок 1.2 – Конфигурирование нового проекта в Android Studio

Далее указываем минимальную версию SDK платформы (рисунок 1.3) – самую раннюю версию Android, которую будет поддерживать приложение. Обратите внимание, что Minimum SDK не может быть выше версии API той мобильной платформы, на которой планируется тестирование и последующее использование приложения.

В появившемся окне Add an activity to Mobile в качестве основы будущего приложения выбираем Empty Activity, в результате чего будет создано приложение с одним Activity.

В форме Customize the Activity (рисунок 1.4) сгенерированы имена файлов класса Activity (Activity Name), графической разметки (Layout Name). Рекомендуется оставить данные имена без изменений, т. к. для запускающего Activity они являются общепринятыми.

После этого Android Studio создает новый проект с указанными характеристиками. Структура проекта (рисунок 1.5) включает: файл AndroidManifest.xml (манифест приложения), папку java (содержит весь код приложения), папку res (используется для файлов-ресурсов различного типа: res/drawable/ – для изобра-

жений (PNG, JPEG и т. д.); res/layout/ – для xml-файлов графической разметки; res/menu/ – для xml-файлов меню; res/values/ – для строковых ресурсов, стилей).

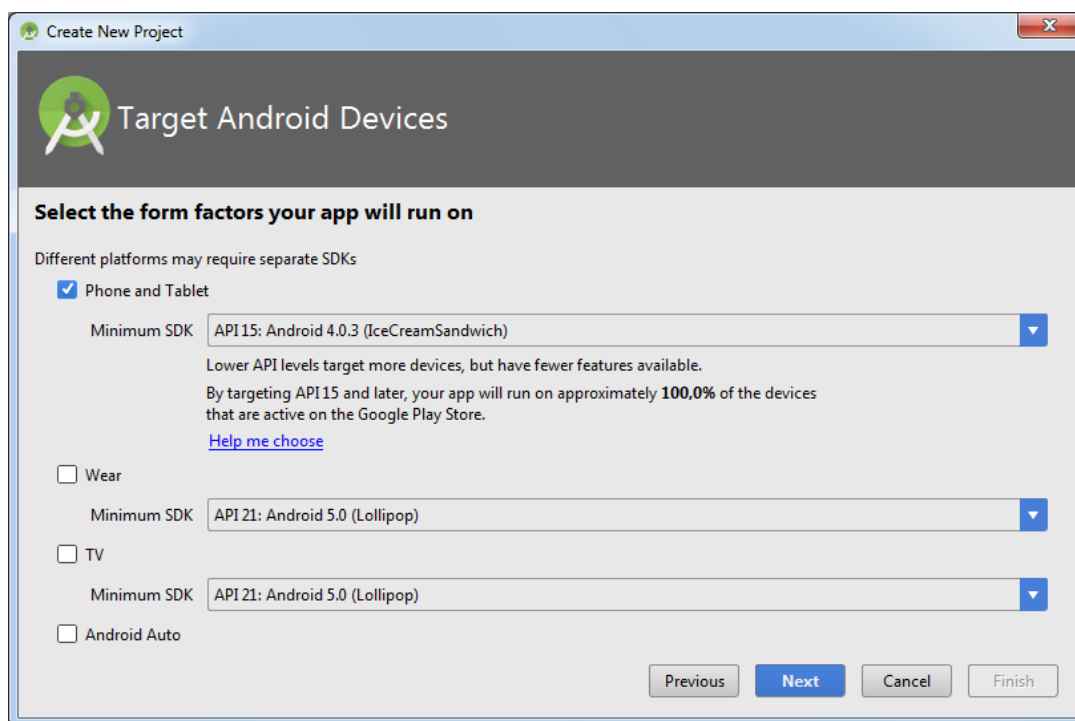


Рисунок 1.3 – Указание минимальной версии SDK для нового проекта в Android Studio

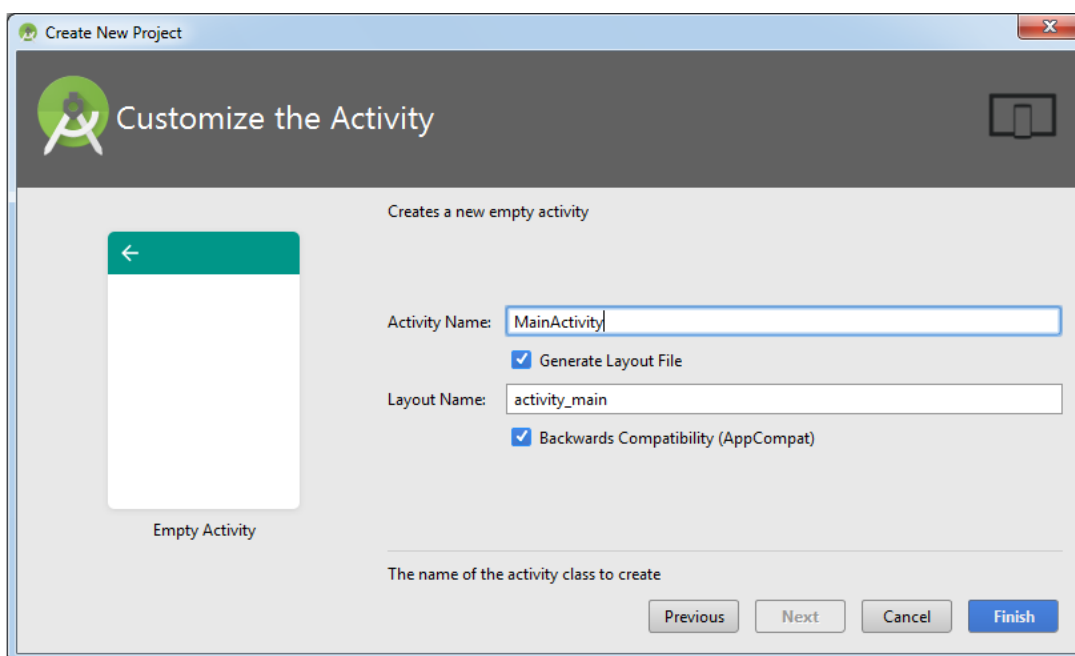


Рисунок 1.4 – Кастомизация имен для нового проекта в Android Studio

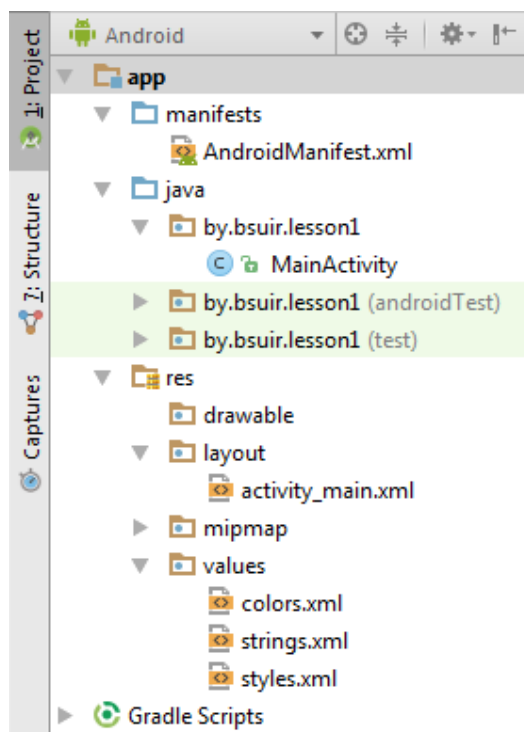


Рисунок 1.5 – Структура проекта в Android Studio

Графическое представление Activity

Основу Android-приложения составляет Activity – рабочее окно. В конкретный момент времени обычно отображается одно Activity и занимает весь экран. Работа с набором окон осуществляется путем переключения различных Activity. В качестве примера можно рассмотреть почтовое приложение: в нем одно Activity – список писем, другое – просмотр письма, третье – настройки ящика.

С программной точки зрения Activity представлено файлом графической разметки (activity_main.xml на рисунке 1.5) и соответствующим java-классом (MainActivity на рисунке 1.5), реализующем запуск данного Activity с требуемым графическим представлением и последующей обработкой событий.

Графическое представление Activity формируется из различных компонентов (кнопка, поле ввода, чекбокс и т. д.), называемых View (рисунок 1.6).

Графическое представление каждого Activity в виде требуемого набора и взаимного расположения View-элементов хранится в xml-файле папки layout. Данный файл графического представления прописывается в соответствующем java-классе. При запуске приложения Activity читает этот файл и отображает его содержимое.

После создания проекта файл activity_main.xml открывается по умолчанию (рисунок 1.7) в режиме конструктора (вкладка Design), помимо которого существует соответствующее xml-описание графического представления (вкладка Text).

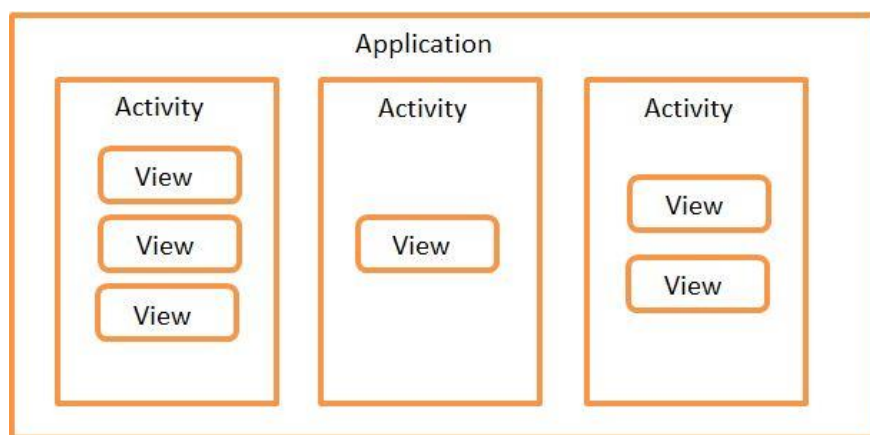


Рисунок 1.6 – Представление приложения Android как набора окон (Activity) с View-компонентами

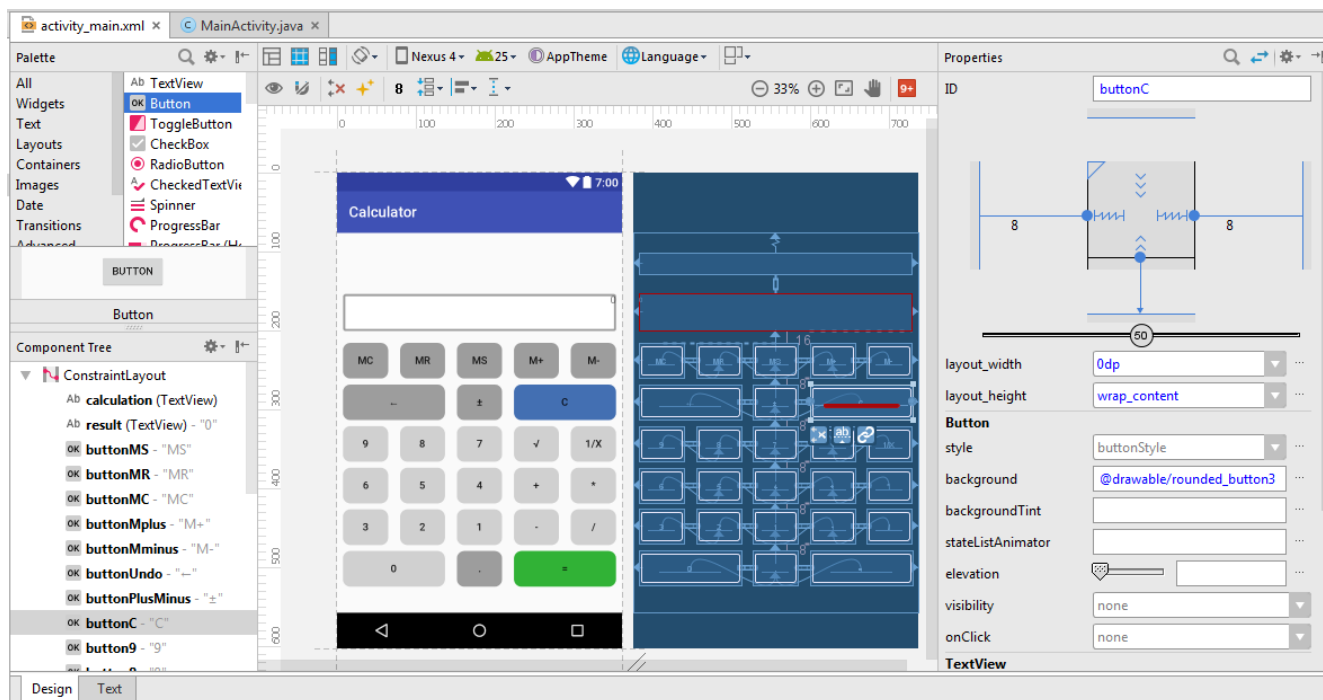


Рисунок 1.7 – Графическое представление Activity. Вкладка Design

В центре вкладки Design по умолчанию расположены два графических редактора, представляющие собой экран мобильного устройства в режиме Design и режиме Blueprint. Режим Blueprint специально разработан для удобства размещения элементов и настройки их взаимосвязей.

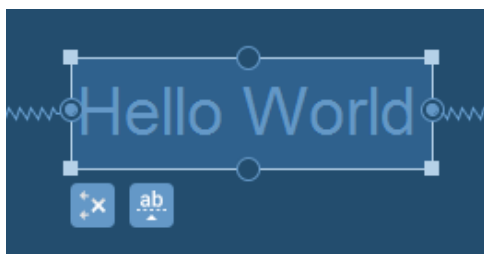
В левой части вкладки Design отображаются окно Palette со списком всех возможных виджетов и окно Component Tree с деревом компонентов, используемых в данном графическом представлении.

В правой частикладки вкладки Design отображается окно Properties с набором свойств для каждого выбранного элемента.

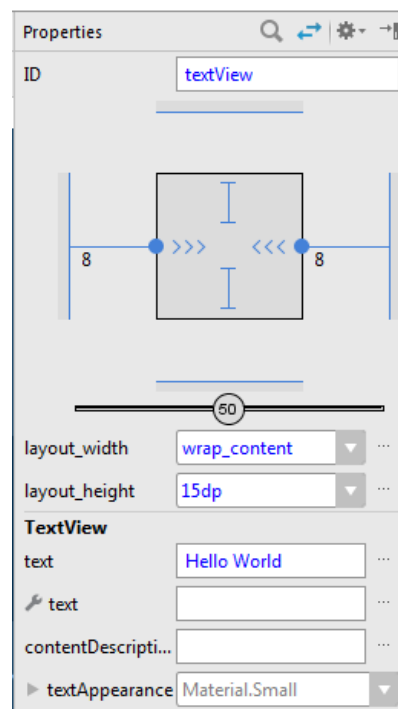
Для размещения View-компонентов используются специальные контейнеры (ViewGroup), называемые Layout. Layout бывают различных типов (LinearLayout, RelativeLayout, FrameLayout, TableLayout, ConstraintLayout и т. д.) и отвечают за то, как будут расположены их дочерние View-компоненты на экране (таблицей, строкой, столбцом). Android Studio по умолчанию использует ConstraintLayout в качестве корневого контейнера для создания разметки экрана и размещения компонентов. Данный контейнер обладает широким спектром возможностей, что позволяет реализовывать сложные взаиморасположения элементов на экране.

Для добавления на экран требуемого компонента необходимо найти его в списке Palette и переместить мышкой на экран в режиме Design или режиме Blueprint. После этого компонент появится на обоих экранах, а также в окне Component Tree.

Квадратные опорные точки в углах компонента (рисунок 1.8, а) позволяют изменять его размеры.



а



б

а – изображение компонента TextView в режиме Blueprint;

б – свойства компонента TextView в окне Properties




Рисунок 1.8 – Пример изображения компонента TextView и его свойств

Круглые опорные точки (рисунок 1.8, а), расположенные по сторонам виджета, позволяют создавать привязки (constraints) к сторонам контейнера или другим компонентам и управлять отступами от краев экрана и других компонентов. Для создания своей собственной привязки необходимо нажать круглую опорную точку на одной из сторон виджета, которую требуется привязать. Далее, не отпуская кнопку мыши, прочертить линию к виджету или краю контейнера. В результате будет создана связь constraint.

В окне Properties (рисунок 1.8, б) находится схематичное изображение компонента, а также указаны его свойства.

Базовым параметром каждого компонента является id – идентификационный номер. Необходимо давать компонентам уникальные и осмысленные имена, чтобы с ними в последующем было удобно работать из java-кода.

Каждый компонент характеризуется шириной (layout_width) и высотой (layout_height). Для установки размера компонента внутри его схематичного изображения в окне Properties необходимо выбрать один из трех возможных типов линий для ширины и высоты:

- линии  означают атрибут wrap content и устанавливают размер элемента по его текущему содержимому;
- линии  означают атрибут fixed и устанавливают фиксированный размер компонента в dp (density-independent pixels, абстрактная единица измерения, позволяющая приложению выглядеть одинаково на различных экранах и разрешениях). Размер компонента можно отредактировать вручную с помощью параметра layout_width или layout_height;
- линии  означают атрибут match constraints и устанавливают максимально допустимый размер элемента.

Переключение между типами линий осуществляется путем нажатия значка, их изображающего.

По сторонам схематичного изображения компонента в окне Properties имеются числа (рисунок 1.9). Они отвечают за атрибут margin (отступы). Если подвести к ним мышку, то появится выпадающий список со значениями для изменения отступа.

Несколько компонентов можно объединять в одну цепь (вертикальную или горизонтальную). Для этого необходимо выделить компоненты и через контекстное меню выбрать опцию Center Horizontally или Center Vertically (рисунок 1.10, а). Рядом с выбранными компонентами появится символ цепи, а между ними будет нарисована связь (рисунок 1.10, б). Если последовательно нажимать значок цепи, то кнопки будут центрироваться с разными стилями.

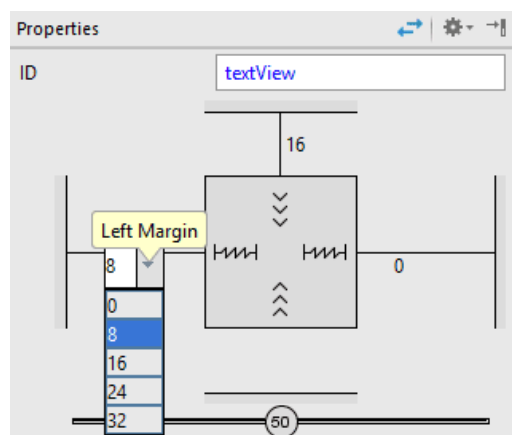
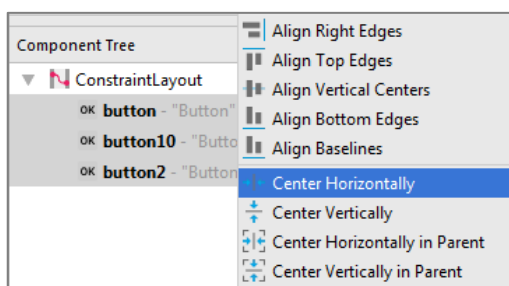
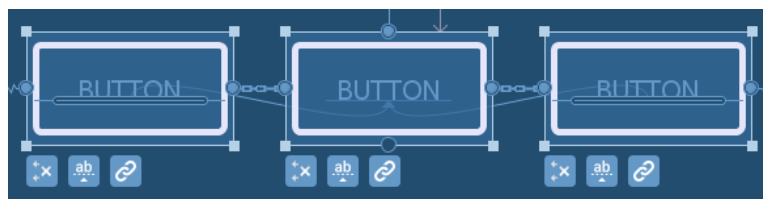


Рисунок 1.9 – Пример установки отступов компонента



а



б

а – объединение компонентов в горизонтальную цепь с помощью опции Center Horizontally; *б* – визуальное отображение компонентов в цепи

Рисунок 1.10 – Пример создания цепи компонентов

Можно присвоить компонентам цепи весовые коэффициенты для пропорционального масштабирования их размеров (при этом соответствующий параметр – высота или ширина – должны иметь атрибут match constraints). Для этих целей используются атрибуты `layout_constraintHorizontal_weight` и `layout_constraintVertical_weight` (их можно найти в расширенном списке Properties).

Закругленный прямоугольник на изображении компонента в режиме Blueprint (рисунок 1.10, б) указывает на базовую линию текста и используется при выравнивании по базовой линии другого компонента [1, 3, 4].

Layout-файл при смене ориентации экрана

По умолчанию layout-файл настроен под вертикальную ориентацию экрана. Однако при повороте смартфона включится горизонтальная ориентация, что может привести к некорректному отображению View-элементов. Для устранения

данной проблемы необходимо создать еще один layout-файл для горизонтальной ориентации экрана. Эта задача решается с помощью опции Create Landscape Variation (рисунок 1.11).

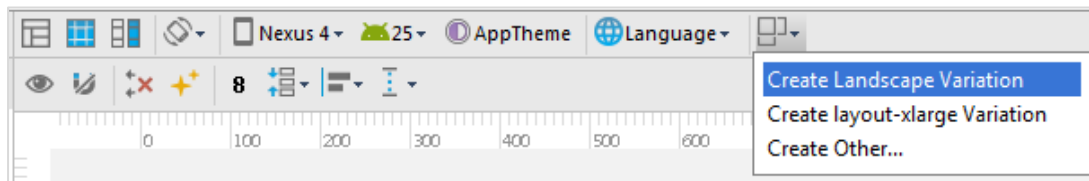


Рисунок 1.11 – Создание графического представления для горизонтальной ориентации экрана

В структуре проекта появится новый xml-файл `activity_main.xml(land)`. При этом текущее содержимое файла `activity_main.xml` будет скопировано в горизонтальное представление `activity_main.xml(land)`. Далее меняем параметры `view`-элементов таким образом, чтобы в горизонтальной ориентации экрана они отображались корректно. При этом `id` элементов не меняем!

Все последующие изменения `activity_main.xml` никак не будут влиять на содержимое `activity_main.xml(land)`, в связи с чем создавать горизонтальное представление целесообразно после размещения всех необходимых компонентов в вертикальном представлении и корректировке `id`-параметров этих компонентов.

При запуске приложения `Activity` прочитает подключенный в коде `layout`-файл и отобразит его содержимое. При этом будет учтена ориентация экрана, и в случае горизонтального расположения автоматически отобразится файл `land`.

Файл `MainActivity.java`. Подключение графического представления к `Activity`

При запуске деятельности система должна получить ссылку на корневой узел дерева разметки, который будет использоваться для прорисовки графического интерфейса на экране мобильного устройства. Для этого в методе `onCreate()` необходимо вызвать метод `setContentView()`.

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

Метод `setContentView(int)` устанавливает содержимое `Activity` из `layout`-файла. Но в качестве аргумента указывается не путь к `layout`-файлу (`res/layout/activity_main.xml`), а константа, которая является `id` файла и хранится в

файле R.java. Имена этих id-констант совпадают с именами файлов ресурсов (без расширений).

Можно создать новый xml-файл в папке res > layout и прописать его вместо activity_main в методе setContentView(int). После запуска приложения отобразится новый файл разметки.

Доступ к элементам экрана из кода

Чтобы обратиться к элементу экрана из кода, необходим его id. Он прописывается в окне Properties либо в xml-коде:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="OK"
    android:id="@+id/btnOK" />
```

Зная id View-элемента, обратиться к нему из кода можно по константе R.id.btnOK. Для этого понадобится метод findViewById:

```
public class MainActivity extends ActionBarActivity{

    private Button btnOK;

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

    private void initViews() {
        // находим View-элементы
        btnOK = (Button) findViewById(R.id.btnOK);
    }
}
```

Следует отметить, что в приведенном фрагменте кода обнаружение View-элементов вынесено в отдельный пользовательский (не являющийся библиотечным) метод initViews() для реализации принципа модульного программирования.

Обработка событий на примере нажатия кнопки

Механизм обработки нажатия кнопки основан на использовании интерфейса `View.OnClickListener` и реализации метода `onClick`, в котором и прописывается логика действий в ответ на нажатие (рисунок 1.11).

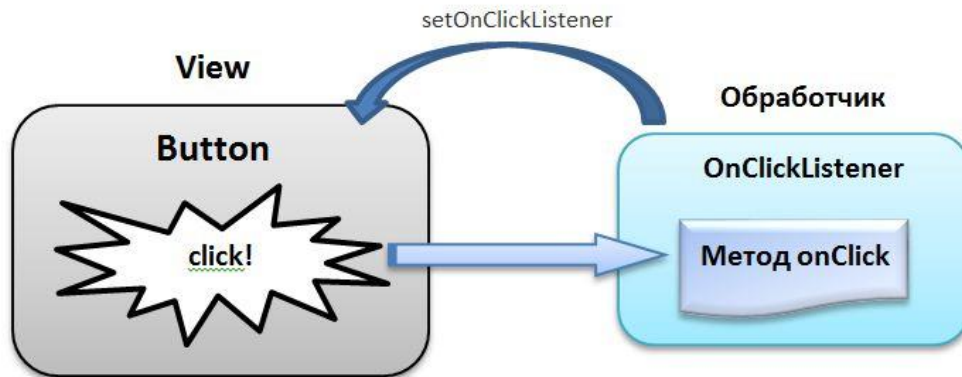


Рисунок 1.11 – Механизм обработки нажатия кнопки на основе интерфейса `View.OnClickListener`

При этом обработчик (его также называют слушателем – `listener`) присваивается кнопке с помощью метода `setOnClickListener`. Когда нажимают кнопку, обработчик реагирует и выполняет код из метода `onClick`. Для реализации данного механизма необходимо выполнить следующие шаги:

- создать обработчик (объект от интерфейса `View.OnClickListener`);
- заполнить метод `onClick` (т. к. в интерфейсе он не реализован, а только заявлен);
- присвоить обработчик кнопке (используем метод `setOnClickListener`).

Система обработки событий готова, а именно, когда нажимают кнопку, обработчик реагирует и выполняет код из метода `onClick` [1, 2].

Пример:

```
OnClickListener listener = new OnClickListener() {
    @Override
    public void onClick(View v) {
        // метод, который будет вызван по нажатию
    }
};
```

```
button.setOnClickListener(listener);
```

Существуют различные способы программной реализации обработки нажатий в зависимости от требуемого набора кнопок одного Activity (с использованием в xml-представлении атрибута `onClick`, своего обработчика для каждого View-элемента, одного обработчика для нескольких View-элементов). Однако наиболее оптимальным является использование Activity в качестве единого обработчика. В данном случае сам класс Activity реализует интерфейс `View.OnClickListener`:

```
public class MainActivity extends ActionBarActivity implements
View.OnClickListener{

    private Button btnOK, btnCancel;

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

    private void initView() {
        // находим View-элементы
        btnOK = (Button) findViewById(R.id.btnOK);
        btnCancel = (Button) findViewById(R.id.btnCancel);
        // подключаем обработчик к кнопкам
        btnOK.setOnClickListener(this);
        btnCancel.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.btnOK:
                btnOK.setText("Hello");
                btnOK.setEnabled(false);
                btnCancel.setEnabled(true);
                break;
            case R.id.btnCancel:
```

```

        btnCancel.setText("Goodbye");
        btnCancel.setEnabled(false);
        btnOK.setEnabled(true);
        break;
    }
}
}

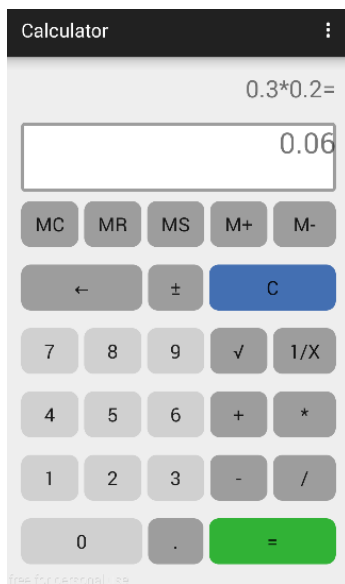
```

Для считывания данных из текстовых полей (например, из поля `et_message`) используется следующий код:

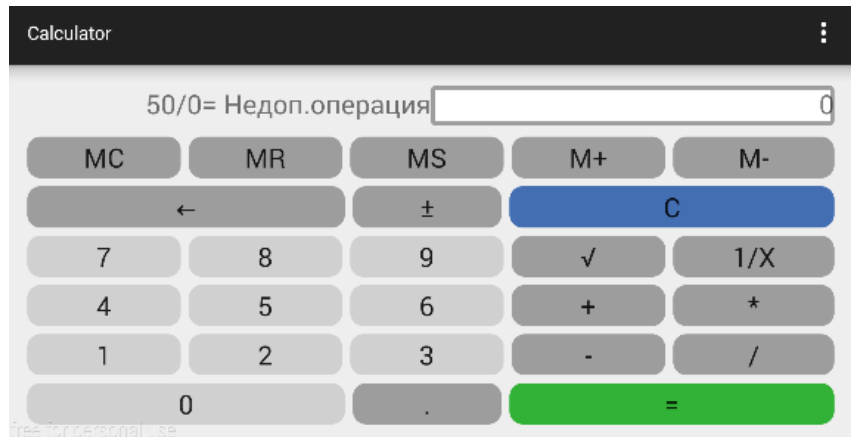
```
String message = et_message.getText().toString();
```

Практическое задание

1. Разработать приложение Calculator с одним Activity.
2. Графическое представление приложения Calculator реализовать с кнопками цифр, математических операций (сложения, вычитания, умножения, деления и др.), получения результата (пример приведен на рисунке 1.12).



a



б

a – вертикальная ориентация экрана; *б* – горизонтальная ориентация экрана

Рисунок 1.12 – Пример работы приложения Калькулятор

3. Предусмотреть возможность ввода дробных чисел через точку (например, 0.5).
4. Предусмотреть возможность ввода отрицательных чисел.
5. Создать графическое представление приложения Calculator для горизонтальной ориентации экрана (пример приведен на рисунке 1.12, б).
6. Программно реализовать обработку нажатий на кнопки с использованием Activity в качестве единого обработчика.
7. В случае деления на 0 выводить вместо результата сообщение о недопустимости операции.
8. Разработать перечень проверок и протестировать приложение Calculator.
9. Продемонстрировать работу приложения Calculator на эмуляторе или реальном устройстве.

Содержание отчета

1. Скриншоты графических представлений приложения Calculator в вертикальной и горизонтальной ориентациях экрана в Android Studio.
2. Код xml-файла графического представления приложения Calculator в вертикальной ориентации экрана.
3. Код xml-файла графического представления приложения Calculator в горизонтальной ориентации экрана.
4. Код java-файла Activity приложения Calculator.

Контрольные вопросы

1. Что такое нативное мобильное приложение, мобильная платформа?
2. Что собой представляет архитектура мобильной платформы Android?
3. Какие основные компоненты Android-приложения Вы знаете?
4. Что собой представляет структура Android-проекта? Что содержит файл конфигурации AndroidManifest.xml, папка java, папка res?
5. Что такое графическое представление Activity?
6. Что такое Layout? Какие существуют виды Layout?
7. Какие параметры (атрибуты) имеют View-элементы?
8. Как создать Layout-файл для работы в горизонтальной ориентации экрана мобильного устройства? В каких случаях это необходимо?
9. Для чего нужны методы setContentView, findViewById?
10. Какие существуют способы обработки событий в Activity?

Лабораторная работа №2

Разработка приложений с несколькими Activity.

Передача данных между Activity

Цель: формирование у студентов знаний о жизненном цикле Activity, навыков создания и вызова нового Activity, передачи данных между Activity, хранения данных с помощью класса SharedPreferences, создания всплывающих сообщений и логирования.

План занятия

1. Изучить теоретические сведения.
2. Выполнить практическое задание по лабораторной работе.
3. Оформить отчет и ответить на контрольные вопросы.

Теоретические сведения

Работа мобильного приложения может сопровождаться возникновением всплывающих сообщений, диалоговых окон, работой с меню и т. д. Рассмотрим реализацию простейшего вспомогательного элемента – всплывающего сообщения.

Всплывающие сообщения

Приложение может показывать всплывающие сообщения с помощью класса Toast:

```
Toast.makeText(context, text , duration).show();
```

Статический метод `makeText` создает View-элемент Toast.

Далее рассмотрим параметры метода:

1) `context` – объект, который предоставляет доступ к базовым функциям приложения; т. к. Activity является подклассом Context, то в качестве объекта от Context используем текущее Activity, а в качестве первого параметра указываем `this`;

2) `text` – текст, который надо вывести;

3) `duration` – продолжительность показа (`Toast.LENGTH_LONG` – длинная (3,5 с), `Toast.LENGTH_SHORT` – короткая (2 с)).

Чтобы Toast отобразился на экране, вызывается метод `show()`.

Пример реализации (рисунок 2.1):

```
Toast.makeText (this, "Show Text Views!", Toast.LENGTH_SHORT).show();
```



a – вывод сообщения «Show Text Views!» внизу окна после нажатия кнопки Show и его размещаемого по умолчанию; *б* – вывод сообщения «Hide Text Views!» в центре окна после нажатия кнопки Hide

Рисунок 2.1 – Пример отображения всплывающих сообщений

По умолчанию всплывающее сообщение отображается в нижней части рабочего окна по центру. Для изменения места расположения всплывающего сообщения используется метод `setGravity`:

```
Toast toast = Toast.makeText(this, "Hide Text Views!",
    Toast.LENGTH_SHORT);
toast.setGravity(Gravity.CENTER, 0, 0);
toast.show();
```

Логирование

При тестировании работы приложения можно отслеживать логи в окне logcat (рисунок 2.2). Логи имеют разные уровни важности: error, warn, info, debug, verbose (по убыванию). Установка Log level в окне logcat приводит к фильтрации логов указанного уровня, а также уровней более высокой важности. Имеется возможность создавать, редактировать и удалять свои фильтры.

Логирование является одним из инструментов разработчика на этапе тестирования приложения. Для этого программно создаются логи, по их выводу в окне logcat отслеживается логика работы приложения, а перед выпуском финальной версии созданные логи из программы удаляются.

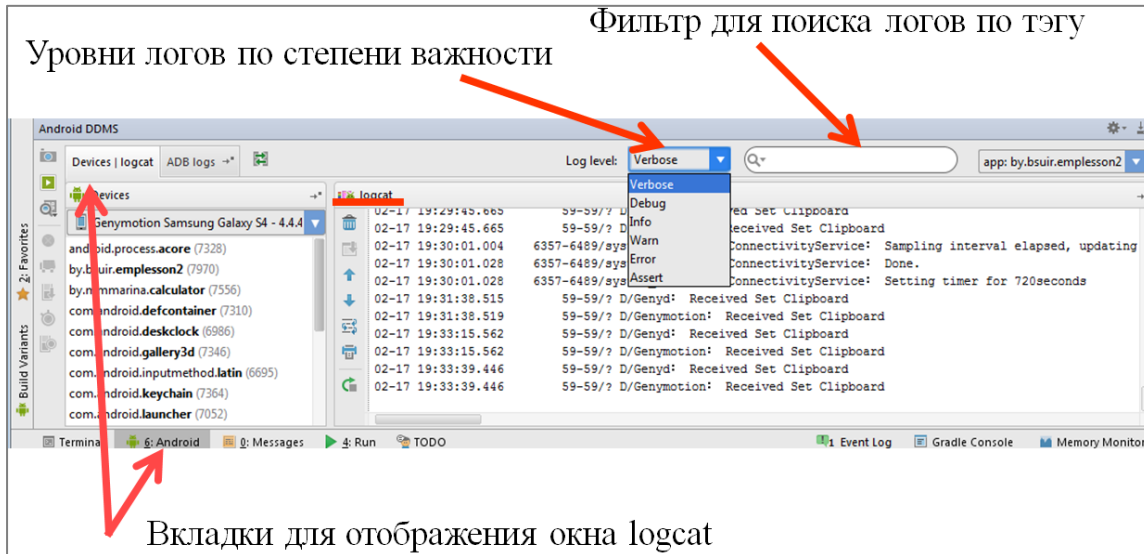


Рисунок 2.2 – Работа с окном logcat

Программно логирование выполняется с помощью класса Log и его методов Log.v() Log.d() Log.i() Log.w() and Log.e(). Названия методов соответствуют уровню логов. Пример синтаксиса методов:

Log.d(tag, text);

Методы требуют на вход тэг и текст сообщения. Тэг – это метка для реализации возможности поиска конкретного лога путем создания собственного фильтра.

Пример реализации логирования в коде:

```
private static final String TAG = "myLogs";
```

```
public void onClick(View v) {
    switch (v.getId()) {
        case R.id.btnShow:
            Log.d(TAG, "Нажата кнопка Show");
    }
}
```



```

        break;
    case R.id.btnHide:
        Log.d(TAG, "Нажата кнопка Hide");
        break;
    }
}

```

Жизненный цикл Activity

Созданное при работе приложения Activity может быть в одном из трех состояний:

1. Resumed: Activity видно на экране, оно находится в фокусе, пользователь может с ним взаимодействовать. Это состояние также иногда называют Running.

2. Paused: Activity не в фокусе, пользователь не может с ним взаимодействовать, но его видно (оно перекрыто другим Activity, которое занимает не весь экран или полупрозрачно).

3. Stopped: Activity не видно (полностью перекрывается другим Activity), соответственно оно не в фокусе и пользователь не может с ним взаимодействовать.

Когда Activity переходит из одного состояния в другое, система автоматически вызывает соответствующие методы, в которые можно добавлять свой код. Методы Activity, которые вызывает система (рисунок 2.3):

- onCreate() – вызывается при первом создании Activity;
- onStart() – вызывается перед тем, как Activity будет видно пользователю;
- onResume() – вызывается перед тем, как будет доступно для активности пользователя (взаимодействие);
- onPause() – вызывается перед тем, как будет показано другое Activity;
- onStop() – вызывается, когда Activity становится не видимым пользователю;
- onDestroy() – вызывается перед тем, как Activity будет уничтожено.

Обратите внимание на то, что сами методы НЕ вызывают смену состояния. Наоборот, смена состояния Activity является триггером, который вызывает эти методы. Тем самым мы можем реагировать на произошедшее событие необходимым образом [1-3, 6-7].

Для того чтобы в ответ на происходящие изменения состояния Activity отработал требуемый код, необходимо переопределить описанные методы жизненного цикла Activity (вызов и вставка в код переопределяемого метода производится с помощью комбинации клавиш Ctrl + O).

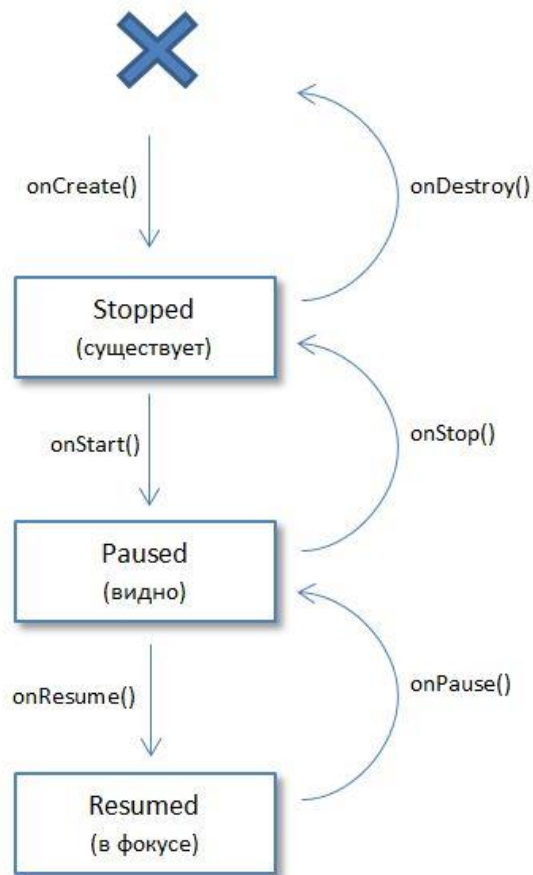


Рисунок 2.3 – Жизненный цикл Activity

Далее приведен фрагмент кода, переопределяющего методы `onStart()` и `onResume()` путем добавления логирования.

```
@Override
protected void onStart() {
    super.onStart(); // метод родительского класса (его нельзя удалять!)
    Log.d(TAG, "MainActivity: onStart"); // добавляем логи
}
@Override
protected void onResume() {
    super.onResume();
    Log.d(TAG, "MainActivity: onResume"); // добавляем логи
}
```

Аналогичным образом можно переопределить все шесть методов, включая onCreate(). Следует отметить, что добавочный код необходимо располагать только после вызова родительского метода (например, super.onStart() для onStart()).

Создание нового Activity

Мобильное приложение, как правило, содержит несколько Activity и реализует переход между ними с передачей данных при необходимости.

Создать новое Activity можно вручную или воспользоваться встроенной функцией добавления нового компонента – Activity – в существующий проект. Далее рассмотрим оба способа.

Алгоритм создания нового Activity вручную включает следующие шаги:

1. Создаем новый xml-файл разметки (например, activity_second.xml).
2. Создаем новый java-класс второго Activity (например, SecondActivity).
3. Прописываем логику SecondActivity.java (наследуемся от суперкласса, подключаем графическую разметку):

```
public class SecondActivity extends ActionBarActivity{
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
    }
}
```

4. Регистрируем новое Activity в манифест-файле (это обязательно):

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

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

Программный способ добавления нового компонента в Android-приложение реализуется с помощью меню File → New или путем нажатия комбинации клавиш Alt + Insert. Далее необходимо заполнить форму кастомизации нового Activity с указанием имен java-файла, xml-файла по аналогии с созданием Activity в новом проекте (см. рисунок 1.4). Запись в файле манифеста будет создана автоматически.

Явный вызов нового Activity

Следующий шаг после создания нового Activity – научиться вызывать это Activity, например в ответ на нажатие одноименной кнопки в исходном Activity (рисунок 2.4).



Рисунок 2.4 – Пример вызова нового Activity в ответ на нажатие одноименной кнопки в исходном Activity

Для того чтобы из одного Activity вызвать другое, существует два способа: явный и неявный вызов.

Реализуем явный вызов нового Activity из основного:

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

Намерение (Intent) – это механизм для описания одной операции (выбрать фотографию, отправить письмо, сделать звонок, запустить браузер и перейти по указанному адресу). В Android-приложениях многие операции работают через намерения. Наиболее распространенный сценарий использования намерения – запуск другой активности в своем приложении.

С помощью класса Intent явно указывают, какое Activity необходимо отобразить (это обычно используется внутри одного приложения):

```
Intent (Context packageContext, Class cls)
```

Context – это объект, который предоставляет доступ к базовым функциям приложения, таким как доступ к ресурсам, к файловой системе, вызов Activity и т. д. Activity является подклассом Context, поэтому в качестве объекта Context можно прописать this.

Class – имя класса, указанное в манифест-файле. Так как при создании записи Activity в манифест-файле было указано имя класса, то это же имя следует указать во втором параметре Intent. Далее, просмотрев манифест-файл, система обнаружит соответствие и покажет соответствующее Activity [1].

Неявный вызов нового Activity

В отличие от явного вызова, основанного на поиске нового Activity по имени класса в манифест-файле, неявный вызов работает с настройками Intent Filter искомого Activity в манифест-файле.

Intent Filter включает ряд параметров (action, data, category), комбинация которых определяет желаемую цель (отправка письма, открытие гиперссылки, редактирование текста, просмотр картинки, звонок по определенному номеру и т. д.).

Для настройки этих параметров в манифест-файле для нового Activity добавляем Intent Filter:

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

Тогда неявно вызвать SecondActivity из MainActivity можно следующим образом:

```
intent = new Intent("android.intent.action.SecondActivity");
startActivity(intent);
```

Параметры объекта Intent из приведенного кода совпадают с условиями фильтра в манифест-файле, и SecondActivity будет вызвано.

Следует отметить, что в случае с неявным вызовом поиск идет уже по всем Activity всех приложений в системе. Если таковых находится несколько, то система предоставляет выбор, какой именно программой необходимо воспользоваться.

Передача данных между Activity

Передача данных между Activity осуществляется с помощью класса Intent. Сохранение данных в Intent реализуется с помощью метода putExtra(), а извлечение – getExtra().

Метод putExtra добавляет к объекту пару: первый параметр – это ключ (имя), второй – значение.

Пример кода из MainActivity.java:

```
Intent intent = new Intent(this, SecondActivity.class);
intent.putExtra("Name", et_Name.getText().toString());
intent.putExtra("Email", et_Email.getText().toString());
startActivity(intent);
```

Метод getExtra извлекает значение по ключу. Пример кода из SecondActivity.java:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_second);
    initView();

    Intent intent = getIntent();
    tv_Name.setText(intent.getStringExtra("Name"));
    tv_Email.setText(intent.getStringExtra("Email"));
}
```

Поместить в Intent можно данные не только типа String. Посредством ссылки <http://developer.android.com/reference/android/content/Intent.html#pubmethods> можно ознакомиться со всеми типами данных, которые принимает на вход метод putExtra.

Чтобы сохранить пользовательский тип данных используется метод putSerializable(), а извлечение – getSerializable() соответственно [1].

Метод startActivityForResult

Запуск другого Activity не обязательно должен быть односторонним. Можно запустить другое Activity и получить обратно результат. Для этого используется метод startActivityForResult() вместо startActivity() (рисунок 2.5).

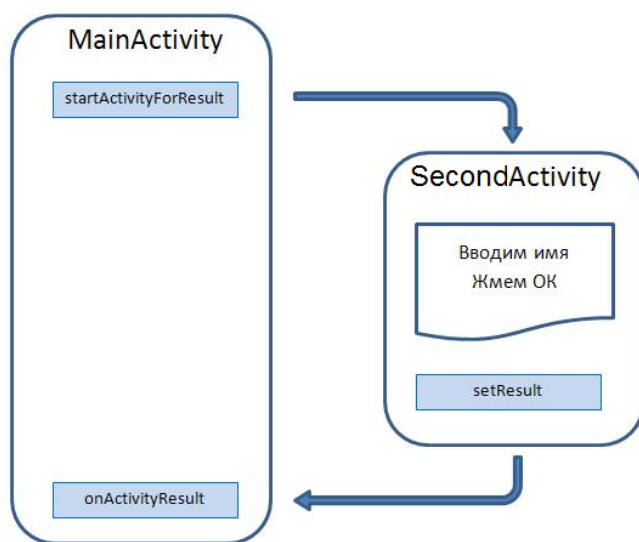


Рисунок 2.5 – Механизм вызова нового Activity и возврат из него с результатом

Алгоритм действий следующий:

1. Вызвать из исходного Activity новое Activity не с помощью метода startActivity, а с помощью startActivityForResult.

Синтаксис метода startActivityForResult:

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

Здесь requestCode – идентификатор для определения, с какого Activity пришел результат.

2. Реализовать в новом Activity метод setResult для указания, какие данные необходимо вернуть в «родительское» Activity и вызвать finish().

Синтаксис метода setResult:

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

Здесь resultCode – код возврата, определяющий, успешно прошел вызов или нет;

intent адресует данные в «родительское» Activity.

3. В исходном Activity в методе onActivityResult прописать логику обработки результата, который придет из нового Activity.

Синтаксис метода onActivityResult:

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

Здесь requestCode – тот же идентификатор, что и в startActivityForResult, по нему определяем, с какого Activity пришел результат;

resultCode – код возврата;

data – Intent, в котором возвращаются данные [1].

Например, необходимо из MainActivity вызвать SecondActivity, в SecondActivity ввести имя пользователя и вернуться в MainActivity, чтобы отобразить это имя в текстовом поле.

MainActivity.java:

```
@Override
```

```
public void onClick(View v) {  
    switch (v.getId()) {  
        case R.id.btnGotoActivity2:  
            Intent intent = new Intent(this, SecondActivity.class);  
            startActivityForResult(intent, 1);  
            break;  
    }  
}
```

```
@Override
```

```
protected void onActivityResult(int requestCode, int resultCode, Intent data){  
    if (data == null) {  
        return;  
    }  
    String name = data.getStringExtra("Name");  
    tv_Name.setText(name);  
}
```


SecondActivity.java:

```
@Override
public void onClick(View view) {
    Intent intent = new Intent();
    intent.putExtra("Name", et_Name.getText().toString());
    setResult(RESULT_OK, intent);
    finish();
}
```

Хранение данных в виде пары «ключ – значение» с помощью класса SharedPreferences

Если имеется относительно небольшая коллекция пар «ключ – значение», которую требуется сохранить, целесообразно использовать SharedPreferences API. Объект SharedPreferences указывает на файл, содержащий пары «ключ – значение» и обеспечивает простые методы для их чтения и записи. Каждый SharedPreferences файл управляется библиотекой и может быть личным или общим. Примером использования SharedPreferences может служить задача сохранения персональных данных пользователя и их восстановления при последующих запусках приложения.

Для реализации задачи хранения данных создадим два пользовательских метода: saveText() – сохранение данных – и loadText() – загрузка данных. Загрузку будем выполнять при запуске приложения (т. е. в методе onCreate), а сохранение – при закрытии приложения (т. е. в методе onDestroy).

Для сохранения данных сначала с помощью метода getPreferences получаем объект sPref класса SharedPreferences, который позволяет работать с данными (читать и писать):

```
sPref = getPreferences(MODE_PRIVATE);
```

Константа MODE_PRIVATE используется для настройки доступа и означает, что после сохранения данные будут доступны только этому приложению.

Далее, чтобы редактировать данные, необходим объект Editor. Получаем его из sPref:

```
SharedPreferences.Editor editor = sPref.edit();
```

В методе putString указываем наименование переменной, т.е. константы SAVED_TEXT, и значение – содержимое поля myEditText. Чтобы данные сохра-

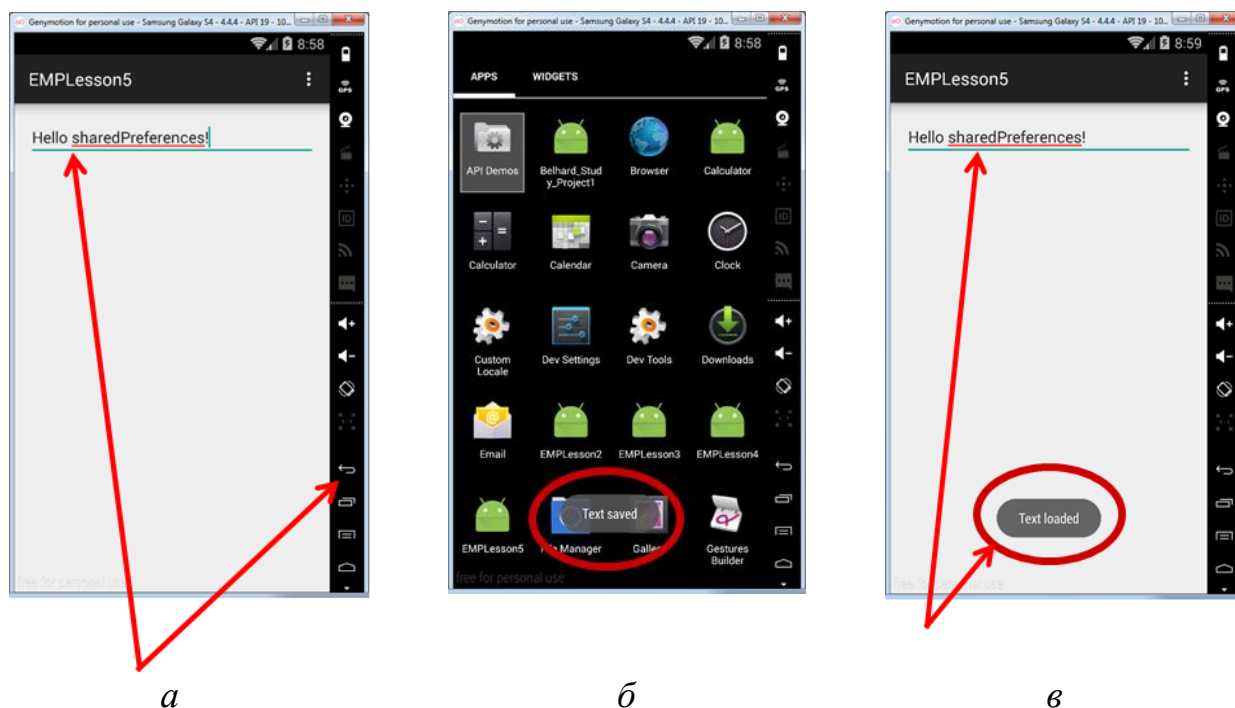
нились, необходимо выполнить commit. И для наглядности выводим сообщение, что данные сохранены:

```
editor.putString(SAVED_TEXT, myEditText.getText().toString());
editor.commit();
Toast.makeText(this, "Text saved", Toast.LENGTH_SHORT).show();
```

Для загрузки данных получаем объект sPref класса SharedPreferences. Чтение осуществляется с помощью метода getString: в параметрах указываем константу и значение по умолчанию (пустая строка). Далее пишем значение в поле ввода myEditText и выводим сообщение, что данные считаны:

```
sPref = getPreferences(MODE_PRIVATE);
String savedText = sPref.getString(SAVED_TEXT, "");
myEditText.setText(savedText);
Toast.makeText(this, "Text loaded", Toast.LENGTH_SHORT).show();
```

Работа приложения продемонстрирована на рисунке 2.6.



а – сохранение данных и выход из приложения; *б* – всплывающее сообщение;
в – приложение с восстановленной информацией

Рисунок 2.6 – Хранение данных с использованием класса SharedPreferences

Полный код приложения приведен ниже:

```
public class MainActivity extends ActionBarActivity {

    private EditText myEditText;
    SharedPreferences sPref;
    final String SAVED_TEXT = "my_saved_text";

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

    private void initView() {
        myEditText = (EditText) findViewById(R.id.myEditText);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        saveText();
    }

    private void saveText() {
        sPref = getPreferences(MODE_PRIVATE);
        SharedPreferences.Editor editor = sPref.edit();
        editor.putString(SAVED_TEXT, myEditText.getText().toString());
        editor.commit();

        Toast.makeText(this, "Text saved", Toast.LENGTH_SHORT).show();
    }

    private void loadText() {
        sPref = getPreferences(MODE_PRIVATE);
        String savedText = sPref.getString(SAVED_TEXT, "");
        myEditText.setText(savedText);
    }
}
```

```

        Toast.makeText(this, "Text loaded", Toast.LENGTH_SHORT).show();
    }
}

```

Практическое задание

1. Разработать приложение Taxi, состоящее из трех Activity (рисунок 2.7).
2. В первом Activity создать три редактируемых текстовых поля (EditText) для ввода пользователем регистрационных данных (телефон, имя и фамилия) и кнопку Registration для запуска второго Activity.
3. При нажатии кнопки Registration выполнить явный вызов второго Activity с передачей данных о пользователе (телефон, имя и фамилия).
4. Во втором Activity создать два текстовых поля (TextView) для вывода переданной информации о пользователе (имя и фамилия, телефон), пустое по умолчанию текстовое поле (TextView) для вывода маршрута движения, кнопку Set path для ввода этого маршрута, кнопку вызова такси Call Taxi (недоступна, пока не введен маршрут движения).
5. При нажатии кнопки Set path выполнить неявный вызов третьего Activity с помощью метода startActivityForResult.
6. В третьем Activity создать шесть редактируемых текстовых полей (EditText) для ввода параметров маршрута движения, кнопку ОК для возврата во второе Activity.
7. При нажатии кнопки ОК реализовать возврат во второе Activity с передачей в качестве результата параметров маршрута движения.
8. После возврата во второе Activity в текстовое поле вывести информацию о маршруте движения и предложение вызвать такси, кнопку вызова такси Call taxi сделать доступной.
9. При нажатии кнопки Call Taxi вывести всплывающее сообщение об успешной отправке такси.
10. Реализовать сохранение регистрационных данных пользователя в исходном Activity с помощью класса SharedPreferences и восстанавливать эту информацию при повторных запусках приложения. При этом название кнопки Registration должно программно меняться на Log in.
11. Вывести в лог очередность вызовов методов жизненного цикла первого, второго и третьего Activity.
12. Сделать вывод на основании логов о жизненном цикле Activity.



a – исходное Activity; *б* – исходное Activity с введенными данными; *в* – второе Activity с выводом данных; *г* – третье Activity для ввода маршрута движения; *д* – возврат во второе Activity; *е* – исходное Activity при повторных запусках

Рисунок 2.7 – Пример работы приложения Taxi

13. Продемонстрировать работу приложения Taxi на эмуляторе или реальном устройстве.

14. Дополнительное задание, предполагающее самостоятельное углубленное освоение материала: реализовать функционал гео-локации для автоматического определения исходной точки маршрута движения и/или Google-карт для обозначения конечной точки маршрута движения пользователя.

Содержание отчета

1. Скриншоты графических представлений первого, второго и третьего Activity в Android Studio, демонстрирующие логику работы приложения Taxi.

2. Код xml-файлов графических представлений первого, второго и третьего Activity приложения Taxi.

3. Код java-файлов первого, второго и третьего Activity приложения Taxi.

4. Результаты вывода в лог очередности вызовов методов жизненного цикла первого, второго и третьего Activity приложения Taxi.

Контрольные вопросы

1. Как с помощью класса Toast создать всплывающее сообщение?

2. В каких случаях необходимо логирование?

3. Что представляет собой окно LogCat? Какие существуют уровни логирования?

4. Как программно реализовать логирование?

5. Как создать новое Activity (опишите работу с java-классом, layout-файлом, файлом конфигурации AndroidManifest.xml)?

6. Для чего используется контекст приложения Context?

7. Для чего используются объекты класса Intent?

8. Как выполнить явный вызов Activity?

9. Как выполнить неявный вызов Activity?

10. Какие состояния предусмотрены жизненным циклом Activity?

11. Какие методы автоматически срабатывают при смене состояния Activity? Как можно использовать эти методы?

12. Как выполняется передача данных с помощью Intent?

13. В каком виде хранятся данные с помощью класса SharedPreferences?

14. В каких случаях целесообразно хранить данные с помощью класса SharedPreferences?

Лабораторная работа №3

Списки. Создание собственного адаптера.

Механизмы обратного вызова

Цель: формирование у студентов знаний и навыков создания кастомизированных списков на основе собственного адаптера, реализации механизмов обратного вызова для отслеживания событий в многофункциональных Android-приложениях.

План занятия

1. Изучить теоретические сведения.
2. Выполнить практическое задание по лабораторной работе.
3. Оформить отчет и ответить на контрольные вопросы.

Теоретические сведения

Для разработки многофункциональных интерфейсов Android-приложений необходимо научиться создавать свои собственные наборы View-элементов в отдельном xml-файле с последующим подключением созданной графики в родительскую разметку.

Создание View-элемента из содержимого layout-файла

Класс `LayoutInflater` используется для создания View-элемента из содержимого отдельного layout-файла. Алгоритм действий в данном случае следующий:

1. Создаем xml-файл с требуемым View-элементом или набором View-элементов (например, `my_view.xml`).
2. В `Activity`, к которому требуется добавить содержимое `my_view.xml`, находим `id` того `layout`, в котором и разместиться View-элемент (например, `lin_layout`).
3. В методе `onCreate` создаем объект класса `LayoutInflater`:

```
LayoutInflater inflater = LayoutInflater.from(this);
```

4. Вызываем из-под объекта `inflater` метод `inflate`:

```
View view = inflater.inflate(R.layout.my_view, lin_layout, true);
```

Синтаксис метода `inflater`:

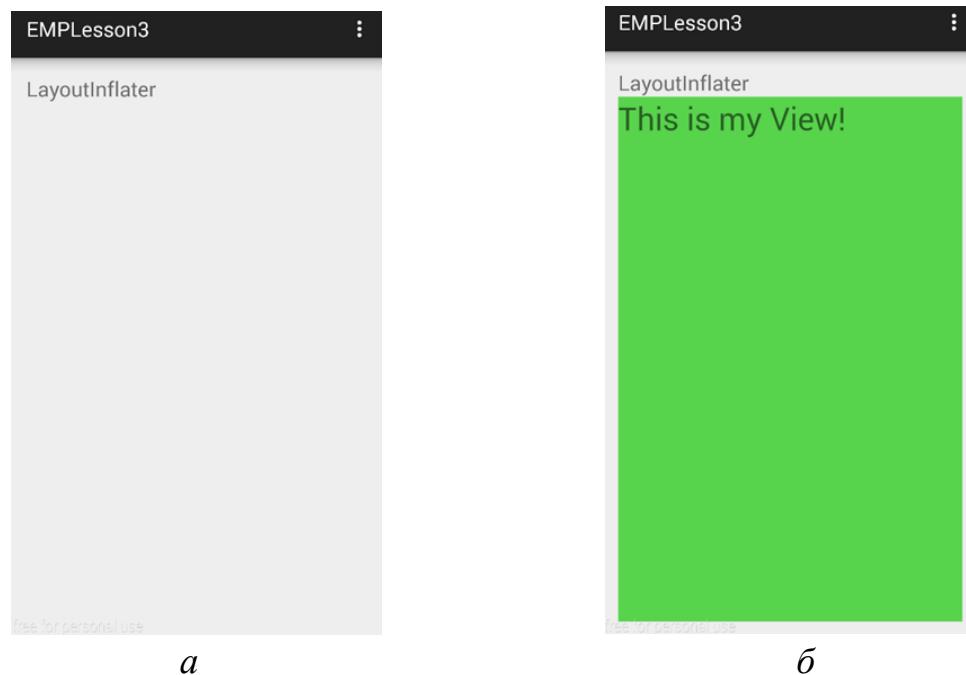
```
public View inflate (int resource, ViewGroup root, boolean attachToRoot);
```

Здесь resource – id layout-файла, который будет использован для создания View;

root – родительский ViewGroup-элемент для создаваемого View;

attachToRoot – параметр, определяющий присоединять ли создаваемый View к root. Чтобы root стал родителем создаваемого View необходимо указать true [1].

Пример полного варианта кода с добавлением View-элемента из содержимого layout-файла как дочернего приведен ниже на рисунке 3.1.



a – первоначальный вид интерфейса; *б* – интерфейс с добавлением дочернего View-элемента из содержимого layout-файла

Рисунок 3.1 – Пример добавлением View-элемента из содержимого layout-файла

```
public class MainActivity extends ActionBarActivity {  
  
    private LinearLayout lin_layout;  
    private LayoutInflater inflater;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {
```



```

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

private void initView() {
    lin_layout = (LinearLayout) findViewById(R.id.lin_layout);

    layoutInflater = LayoutInflater.from(this);
    View view = layoutInflater.inflate(R.layout.my_view, lin_layout, true);
}
}

```

Список ListView.

Список ListView содержит однотипные по дизайну пункты и логику обработки взаимодействия с элементами этих пунктов (рисунок 3.2). Каждый пункт списка представляет собой ViewGroup, создаваемый в отдельном layout-файле.



Рисунок 3.2 – Пример реализации списка ListView

Стратегия создания списка следующая:

1. Создаем в папке layout xml-файл с графическим отображением одного пункта списка. Для корректного отображения набора пунктов в результирующем списке в качестве layout_height указываем wrap_content или высоту в dp.
2. Создаем набор данных для последующего наполнения списка.
3. Программируем свой собственный адаптер. Адаптер – структурный шаблон проектирования, предназначенный для создания класса-оболочки с требуемым интерфейсом.

Адаптеру назначаем набор данных для наполнения списка и layout-ресурс с визуальным представлением одного пункта списка.

4. Присваиваем адаптер списку ListView. Список при построении запрашивает у адаптера пункты, адаптер их создает (используя данные и layout-файл) и возвращает списку. В итоге мы видим готовый список.

Рассмотрим подробно изложенную стратегию. В качестве примера разработаем магазин товаров (см. рисунок 3.2).

Создадим модель данных – класс Good с полями int id (номер товара), String name (наименование товара), boolean check (флаг, отображающий выбор товара пользователем), конструктором класса и методами getId(), getName(), isCheck(), setId(int id), setName(String name), setCheck(boolean check).

Разработаем графическое представление одного пункта списка – файл item_good.xml, содержащий два TextView и один CheckBox для отображения параметров товара.

В activity_main.xml в корневой layout добавляем ListView с размерами wrap content.

Создаем класс GoodsAdapter для реализации кастомизированного адаптера.

В MainActivity.java находим listView по id; создаем динамический массив ArrayList<Good> arr_goods и имитируем интернет-магазин путем генерации объектов класса Good. Создаем объект goodsAdapter от класса GoodsAdapter: для этого в конструктор класса передаем параметр context (т. е. this) и коллекцию объектов arr_goods. Далее присваиваем goodsAdapter списку с помощью метода setAdapter.

```
public class MainActivity extends ActionBarActivity {  
  
    private ListView listView;  
    private ArrayList<Good> arr_goods = new ArrayList<Good>();  
    private final int SIZE_OF_ARR = 25;  
    private GoodsAdapter goodsAdapter;
```

```

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

private void initView() {
    listView = (ListView) findViewById(R.id.listView);
}

private void createMyListView() {
    fillData();
    goodsAdapter = new GoodsAdapter(this, arr_goods);
    listView.setAdapter(goodsAdapter);
}

private void fillData(){
    int i=0;
    while (i<SIZE_OF_ARR) {
        i++;
        arr_goods.add(new Good(i, " " + "My good №" + i, false));
    }
}
}

```

Рассмотрим реализацию класса GoodsAdapter. Наследуемся от базового адаптера BaseAdapter. Для обработки событий от чекбоксов реализуем интерфейс CompoundButton.OnCheckedChangeListener.

```

public class GoodsAdapter extends BaseAdapter implements CompoundBut-
ton.OnCheckedChangeListener{

    private Context context;
    private ArrayList<Good> arr_goods_adapter;

```

```

private LayoutInflater inflater;

public GoodsAdapter(Context context, ArrayList<Good> arr_goods_adapter) {
    this.context = context;
    this.arr_goods_adapter = arr_goods_adapter;
    this.inflater = LayoutInflater.from(context);
}

// количество элементов
@Override
public int getCount() {
    return arr_goods_adapter.size();
}

// элемент по позиции
@Override
public Object getItem(int position) {
    return arr_goods_adapter.get(position);
}

// id по позиции
@Override
public long getItemId(int position) {
    return position;
}

// пункт списка
@Override
public View getView(int position, View convertView, ViewGroup viewGroup) {
    View view = convertView;
    if (view == null) {
        view = inflater.inflate(R.layout.item_good, null, false);
    }

    Good good_temp = arr_goods_adapter.get(position);

    TextView tv_goodId = (TextView) view.findViewById(R.id.tv_goodId);

```

```
tv_goodId.setText(Integer.toString(good_temp.getId()));
```

```
TextView tv_goodName = (TextView) view.findViewById(R.id.tv_goodName);  
tv_goodName.setText(good_temp.getName());
```

```
CheckBox cb_good = (CheckBox) view.findViewById(R.id.cb_good);  
cb_good.setChecked(good_temp.isCheck());  
cb_good.setTag(position);  
cb_good.setOnCheckedChangeListener(this);
```

```
return view;  
}
```

```
@Override
```

```
public void onCheckedChanged(CompoundButton compoundButton, boolean  
isChecked) {  
    if (compoundButton.isShown()) {  
        int i = (int) compoundButton.getTag();  
        arr_goods_adapter.get(i).setCheck(isChecked);  
        notifyDataSetChanged();  
    }  
}  
}
```

В конструкторе заполняем внутренние переменные и получаем `LayoutInflater` для работы с `layout`-ресурсами. В `arr_goods_adapter` хранится список товаров.

Методы, отмеченные аннотацией `@Override`, необходимо реализовать при наследовании `BaseAdapter`. Эти методы используются списком и должны работать корректно.

Метод `getCount` должен возвращать количество элементов списка (в текущем примере это количество товаров).

Метод `getItem` должен возвращать элемент по указанной позиции. Используя позицию, получаем конкретный элемент из `arr_goods_adapter`.

Метод `getItemId` должен возвращать `id` элемента (возвращаем текущую позицию).

Метод `getView` должен возвращать `View` пункта списка. Для этого ранее создавался `layout-ресурс R.layout.item_good`. В этом методе необходимо из `R.layout.item_good` создать `View`, заполнить его данными и отдать списку. Но перед тем как создавать, пробуем использовать `convertView`, который идет на вход метода. Это уже созданное ранее `View`, но неиспользуемое в данный момент. Например, при прокрутке списка, часть пунктов уходит за экран и их уже не надо прорисовывать. `View` из этих «невидимых» пунктов используются для воссоздания графики и обновления данных. Это значительно ускоряет работу приложения, т. к. не надо прорисовывать `inflate` лишний раз.

Если же `convertView` не поступил (`null`), то создаем сами `View`.

Далее заполняем параметры товара, чекбоксу присваиваем обработчик, сохраняем в `Tag` позицию элемента. `Tag` – это подобие `Object`-хранилища у каждого `View`, куда можно поместить требуемые данные. В нашем случае для каждого чекбокса помещаем в его `Tag` номер позиции пункта списка. Далее в обработчике чекбокса будет возможность этот номер позиции извлечь и определить, в каком пункте списка был нажат чекбокс.

В итоге, метод `getView` возвращает списку полностью заполненное `view`, и список его отобразит как очередной пункт.

Обработчик для чекбоксов `onCheckedChanged`. При нажатии на чекбокс в списке он срабатывает, читает из `Tag` позицию пункта списка и помечает соответствующий товар как положенный в корзину. Без этого обработчика не работало бы помещение товаров в корзину, а на экране значения чекбоксов в списке терялись бы при прокрутке, потому что пункты списка пересоздаются, если они уходят «за экран» и снова появляются. Это пересоздание обеспечивает метод `getView`, а он для заполнения `View` берет данные из товаров. Значит, при нажатии на чекбокс, обязательно надо сохранить в данных о товаре информацию о том, что он теперь в корзине. Метод `notifyDataSetChanged()` уведомляет список об изменении данных для обновления списка на экране [1-3].

Header и Footer в списках

`Header` и `Footer` – это `View`-элементы, которые могут быть добавлены к списку сверху и снизу. Для этого необходимо создать графические представления `header_mygoods.xml` и `footer_mygoods.xml`, преобразовать их в `View`-элементы и предоставить списку с помощью методов `addHeader` или `addFooter`. Обязательным условием отображения `Header` и `Footer` является их добавление к списку до того, как присваивается адаптер.

Модифицируем код `MainActivity.java`:

```

public class MainActivity extends ActionBarActivity implements
View.OnClickListener {

    // добавляем новые переменные класса
    private LayoutInflater inflater;
    private View view_header, view_footer;
    private Button btnShow;
    private TextView tv_count;

    // добавляем Header и Footer
    private void createMyListView() {
        fillData();
        goodsAdapter = new GoodsAdapter(this, arr_goods, this);

        inflater = LayoutInflater.from(this);
        view_header = inflater.inflate(R.layout.header_mygoods, null);
        view_footer = inflater.inflate(R.layout.footer_mygoods, null);
        btnShow = (Button) view_footer.findViewById(R.id.btnShow);
        btnShow.setOnClickListener(this);
        tv_count = (TextView) view_footer.findViewById(R.id.tv_count);

        listView.addHeaderView(view_header);
        listView.addFooterView(view_footer);

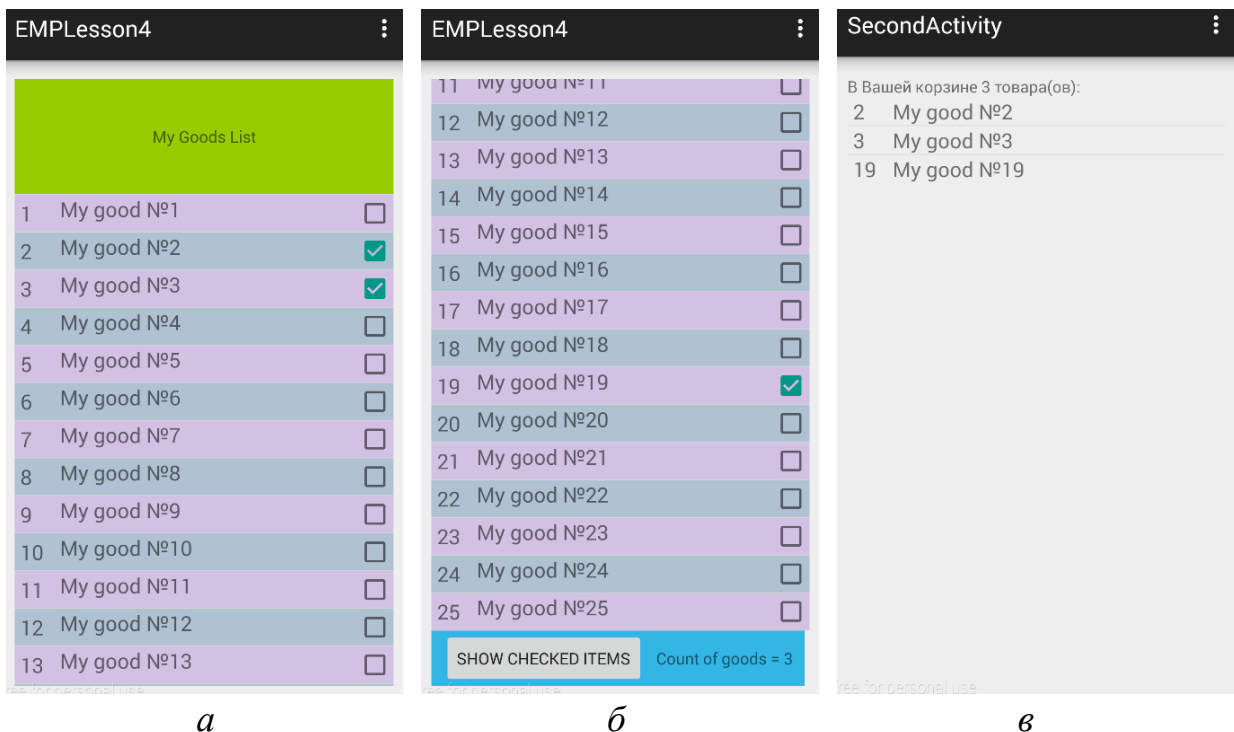
        listView.setAdapter(goodsAdapter);
    }
}

```

Пример реализации списка с Header приведен на рисунке 3.3, а, с Footer – на рисунке 3.3, б.

Механизмы обратного вызова для обработки событий в Android-приложениях

В продолжение примера со списком товаров магазина поставим следующую задачу: отмеченные товары должны сохраняться в отдельном динамическом массиве для последующего отображения корзины товаров пользователя в новом Activity (рисунок 3.3, в).



a – список с Header; *б* – список с Footer; *в* – корзина товаров

Рисунок 3.3 – Пример реализации списка ListView

Особенностью реализации данной задачи является необходимость обработки события, наступление которого отслеживает класс GoodsAdapter, в классе MainActivity.

Простейший механизм callback (обратный вызов функции) позволяет вызывать из одного класса метод другого класса. Для этого достаточно иметь объект от класса, что дает полный доступ ко всем его методам.

В частности, в GoodsAdapter добавляем логику формирования динамического массива выбранных товаров arr_checked_goods_adapter и создаем метод getCheckedGoods(), возвращающий данный массив.

GoodsAdapter.java:

```
private ArrayList<Good> arr_checked_goods_adapter = new ArrayList<Good>();

@Override
public void onCheckedChanged(CompoundButton compoundButton, boolean isChecked) {
```



```

        if (compoundButton.isShown()) {
            int i = (int) compoundButton.getTag();
            arr_goods_adapter.get(i).setCheck(isChecked);
            notifyDataSetChanged();

            if(isChecked){
                arr_checked_goods_adapter.add(arr_goods_adapter.get(i));
            }else {
                arr_checked_goods_adapter.remove(arr_goods_adapter.get(i));
            }
        }
    }

    public ArrayList<Good> getCheckedGoods() {
        return arr_checked_goods_adapter;
    }
}

```

Тогда в MainActivity реализуем логику обработки нажатия кнопки Show checked items.

MainActivity.java:

```

private ArrayList<Good> arr_checked_goods = new ArrayList<Good>();

@Override
public void onClick(View view) {
    arr_checked_goods = goodsAdapter.getCheckedGoods();

    Intent intent = new Intent(this, SecondActivity.class);
    intent.putParcelableArrayListExtra("MyList", arr_checked_goods);
    startActivity(intent);
}

```

Усложним задачу: количество выбранных товаров должно динамически изменяться в TextView, размещенном в Footer списка (см. рисунок 3.3, б). В этом случае потребуется использовать интерфейс в качестве прослойки для отслеживания событий. Такой подход является распространенным и подробно описан в

шаблоне проектирования Наблюдатель (Observer). Для этого класс реализует специально созданный интерфейс; тем самым у класса появляется механизм, который позволяет получать экземпляру объекта этого класса оповещения от объектов других классов об изменении их состояния, тем самым наблюдая за ними.

Создаем свой собственный интерфейс:

```
public interface OnChangeListener {  
  
    public void onDataChanged();  
  
}
```

В MainActivity.java имплементируем данный интерфейс, а в методе onDataChanged() запрашиваем размер массива с выбранными пользователем товарами и выводим это число в соответствующий TextView.

```
public class MainActivity extends ActionBarActivity implements OnChangeListener, View.OnClickListener {  
  
    @Override  
    public void onDataChanged() {  
        int size = goodsAdapter.getCheckedGoods().size();  
        tv_count.setText("Count of goods = " + size + "");  
    }  
}
```

В конструктор GoodsAdapter.java необходимо передать объект от интерфейса OnChangeListener. Так как MainActivity.java имплементирует указанный интерфейс, то в качестве объекта от интерфейса OnChangeListener выступает собственно MainActivity.java, т. е. передаем this:

```
goodsAdapter = new GoodsAdapter(this, arr_goods, this);
```

Изменения в GoodsAdapter.java следующие: добавляем в поля класса и, соответственно, конструктор, объект от интерфейса OnChangeListener: onChangeListener. В методе обработки нажатий на чекбоксы вызываем из-под объекта onChangeListener метод onDataChanged().

GoodsAdapter.java:

```
private OnChangeListener onChangeListener;

public GoodsAdapter(Context context, ArrayList<Good> arr_goods_adapter,
OnChangeListener onChangeListener) {
    this.context = context;
    this.arr_goods_adapter = arr_goods_adapter;
    this.layoutInflater = LayoutInflater.from(context);
    this.onChangeListener = onChangeListener;
}

@Override
public void onCheckedChanged(CompoundButton compoundButton, boolean isChecked) {
    if (compoundButton.isShown()) {
        int i = (int) compoundButton.getTag();
        arr_goods_adapter.get(i).setCheck(isChecked);
        notifyDataSetChanged();

        if(isChecked){
            arr_checked_goods_adapter.add(arr_goods_adapter.get(i));
        }else {
            arr_checked_goods_adapter.remove(arr_goods_adapter.get(i));
        }
        onChangeListener.onDataChanged();
    }
}
```

Таким образом, при каждом нажатии на чекбокс любого товара автоматически сработает реализация метода `onDataChanged()` в `MainActivity` и произойдет перерасчет количества выбранных пользователем товаров.

Передаем Parcelable-объекты с помощью Intent

Для передачи динамического массива объектов между Activity используются следующие методы:

```
// для отправки данных
intent.putParcelableArrayListExtra("MyList", arr_checked_goods);
```

```
// для извлечения данных
arr_checked_goods = getIntent().getParcelableArrayListExtra("MyList");
```

При этом в классе Good необходимо реализовать интерфейс Parcelable:

```
public class Good implements Parcelable{
    private int id;
    private String name;
    private boolean check;

    // обычный конструктор
    public Good(int id, String name, boolean check){
        this.id = id;
        this.name = name;
        this.check = check;
    }

    @Override
    public int describeContents() {
        return 0;
    }

    // упаковываем объект в Parcel
    @Override
    public void writeToParcel(Parcel parcel, int i) {
        parcel.writeInt(id);
        parcel.writeString(name);
    }

    public static final Parcelable.Creator<Good> CREATOR = new
    Parcelable.Creator<Good>() {
        // распаковываем объект из Parcel
        public Good createFromParcel(Parcel in) {
            return new Good(in);
        }
    }
}
```

```

    public Good[] newArray(int size) {
        return new Good[size];
    }
};

// конструктор, считывающий данные из Parcel
private Good(Parcel parcel) {
    id = parcel.readInt();
    name = parcel.readString();
    check = false;
}
}

```

В методе writeToParcel в объект Parcel упаковывают поля объекта Good.

Creator<MyObject> используется для создания экземпляра Good и заполнения его данными из Parcel. Для этого используется его метод createFromParcel, который необходимо реализовать. На вход поступает Parcel, а вернуть нужно готовый Good. В нашем примере используется конструктор Good(Parcel parcel), который реализован отдельно.

Конструктор Good(Parcel parcel) принимает на вход Parcel и заполняет объект Good данными из Parcel.

Итоговая структура проекта приведена на рисунке 3.4.

Практическое задание

1. Разработать приложение MiniShop, состоящее из двух Activity (см. рисунки 3.3, 3.4).
2. В первом Activity создать список ListView с Header и Footer.
3. В Footer разместить текстовое поле (TextView) для ввода количества активированных пользователем товаров, кнопку Show Checked Items для перехода в корзину товаров.
4. Реализовать кастомизированный список ListView с помощью собственного адаптера, наследующего класс BaseAdapter.
5. В каждом пункте списка отобразить следующую информацию о товаре: идентификационный номер, название, стоимость, чекбокс для возможности выбора товара пользователем.

6. В текстовом поле (TextView) Footer списка динамически отображать общее текущее количество активированных товаров.
7. При нажатии кнопки Show Checked Items реализовать переход во второе Activity с корзиной товаров.
8. Корзину товаров реализовать в виде нового кастомизированного списка с выбранными товарами.
9. Продемонстрировать работу приложения MiniShop на эмуляторе или реальном устройстве.

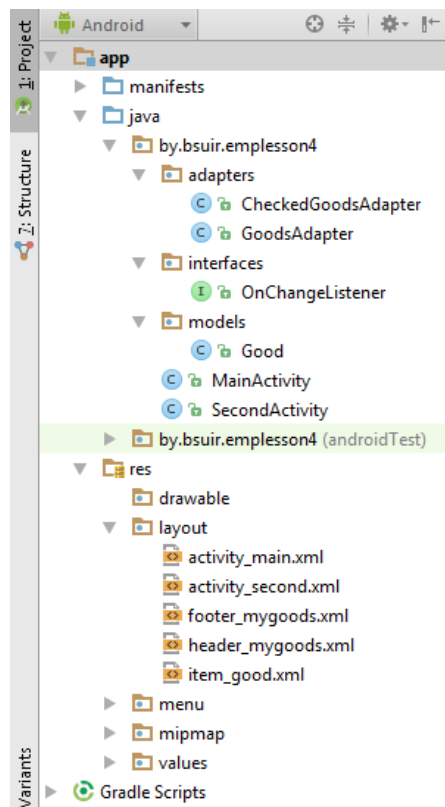


Рисунок 3.4 – Итоговая структура проекта MiniShop

Содержание отчета

1. Скриншоты графических представлений первого и второго Activity в Android Studio, демонстрирующие логику работы приложения MiniShop.
2. Код xml-файлов графических представлений, используемых в приложении MiniShop.
3. Код java-файлов приложения MiniShop, включая классы Activity, собственные интерфейсы, адаптеры, модели данных.

Контрольные вопросы

1. Как создать View-элемент из содержимого layout-файла? В каких случаях это необходимо?
2. Как создать и обеспечить работу списка ListView?
3. Как реализовать собственный кастомизированный список?
4. Что собой представляет и для чего нужен адаптер в Android-приложениях?
5. Какие методы класса BaseAdapter необходимо переопределить при создании кастомизированных списков?
6. Для чего нужен метод getView в адаптере?
7. Что собой представляет и как реализовать Header в списках?
8. Что собой представляет и как реализовать Footer в списках?
9. Какие Вы знаете механизмы обратного вызова для обработки событий в Android-приложениях?
10. Как передать динамический массив объектов из одного Activity в другое с помощью Intent?

Лабораторная работа №4

Фрагменты. ViewPager.

Хранение информации в базе данных SQLite

Цель: формирование у студентов знаний и навыков работы с фрагментами, использования View Pager для перелистывания фрагментов, хранения информации в базе данных SQLite.

План занятия

1. Изучить теоретические сведения.
2. Выполнить практическое задание по лабораторной работе.
3. Оформить отчет и ответить на контрольные вопросы.

Теоретические сведения

Базовые сведения о фрагментах

Устройства с мобильной платформой Android характеризуются многообразием размеров и разрешений экрана, т. е. высокой степенью фрагментации. Если для смартфонов с небольшими экранами реализация Activity выглядит приемлемо, то на больших экранах (например, планшетах) остается незадействованное пространство. В связи с этим начиная с API 11 в Android были добавлены фрагменты, чтобы разработчики могли создавать более гибкие пользовательские интерфейсы на больших экранах (рисунок 4.1) [4].

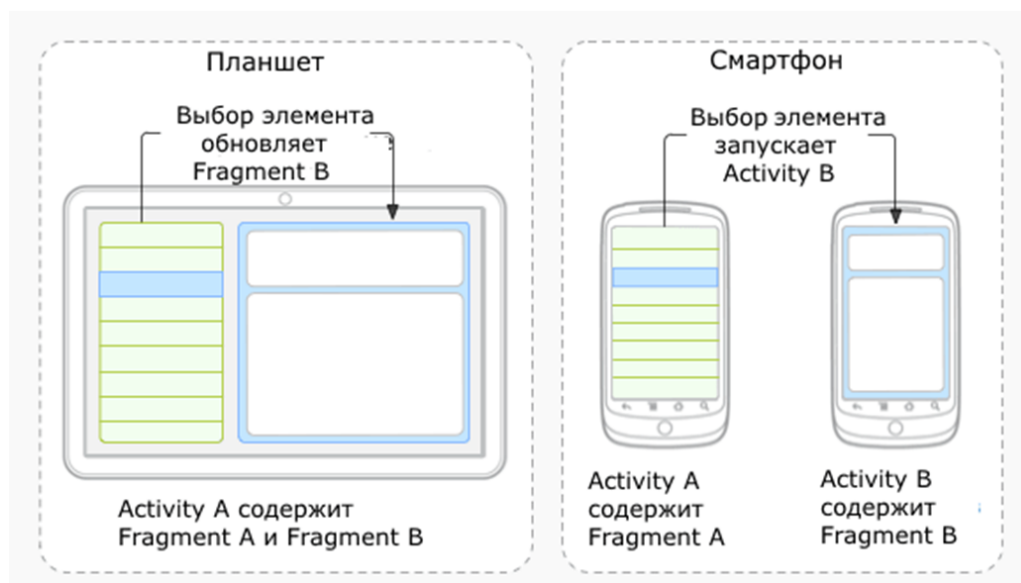


Рисунок 4.1 – Пример реализации концепции фрагментов на Android-устройствах различных размеров

Фрагмент существует в контексте Activity и имеет свой жизненный цикл, вне Activity он существовать не может. Activity может иметь несколько фрагментов.

Алгоритм создания и подключения фрагмента к Activity следующий:

1. В отдельном xml-файле создаем графическое представление фрагмента (например, fragment1.xml).
2. Создаем в папке java класс Fragment1.java. Класс фрагмента должен наследоваться от класса Fragment:

```
public class Fragment1 extends Fragment
```

3. Переопределяем метод onCreateView для подключения разметки фрагмента:

```
public View onCreateView(LayoutInflater inflater, ViewGroup container,  
Bundle savedInstanceState) {  
    View view = inflater.inflate(R.layout.fragment1, container, false);  
    return view;  
}
```

При необходимости в методе onCreateView можно находить View-элементы фрагмента, устанавливать обработчики для этих элементов и реализовывать логику обработки событий подобно Activity.

4. Добавляем фрагмент в Activity одним из двух способов: статическим или динамическим.

Статическое добавление фрагмента

В activity_main.xml фрагмент добавляется как элемент <fragment> (рисунок 4.2). Для каждого фрагмента должны быть установлены высота, ширина, id, а также имя. В качестве имени устанавливается полное имя класса с учетом пакета:

```
<fragment  
    android:layout_width="match_parent "  
    android:layout_height="match_parent"  
    android:name="by.bsuir.emplesson7.fragments.Fragment1"  
    android:id="@+id/fragment1"/>
```

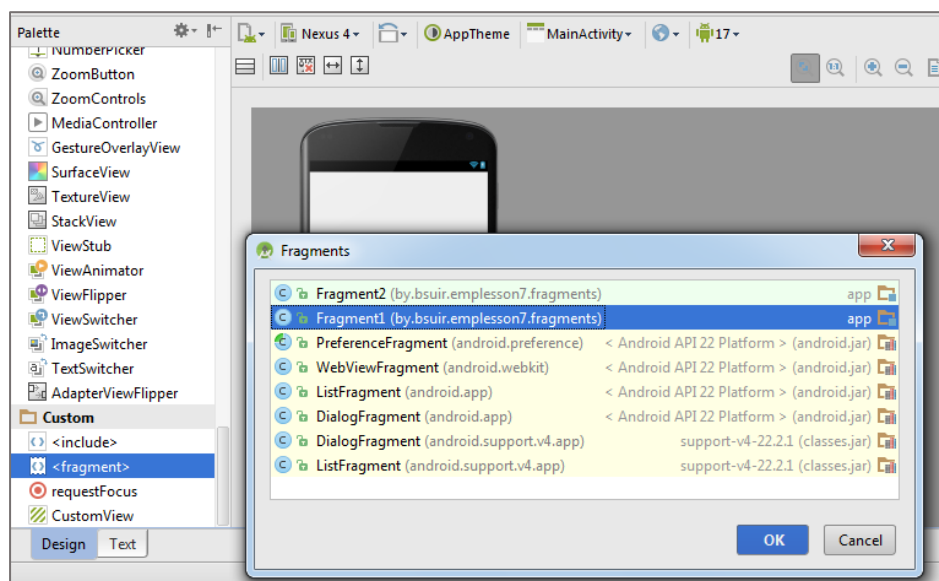


Рисунок 4.2 – Статическое добавление фрагмента к Activity

Кроме фрагмента можно добавить в разметку activity_main.xml другие элементы или фрагменты.

Код класса MainActivity остается тем же, что и при обычном создании проекта.

Динамическое добавление фрагмента

К activity_main.xml добавляем FrameLayout, который впоследствии будет служить контейнером для размещения фрагмента из кода приложения:

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

Добавляем фрагмент в сформированный контейнер непосредственно в коде приложения:

MainActivity.java:

```
private Fragment1 fragment1;
private FragmentManager fragmentManager;
private FragmentTransaction fragmentTransaction;
```

```

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    initView();
    initFragments();
}

private void initFragments() {
    fragment1 = new Fragment1();
    fragmentManager = getSupportFragmentManager;
    fragmentTransaction = fragmentManager.beginTransaction();
    fragmentTransaction.add(R.id.fragment_container, fragment1);
    fragmentTransaction.commit();
}

```

Класс `FragmentManager` служит для управления фрагментами. Чтобы получить его, следует вызвать метод `getFragmentManager()` из кода операции.

Класс `FragmentTransaction` позволяет выполнять следующие транзакции: `add()` – добавление фрагмента; `remove()` – удаление фрагмента; `replace()` – замена фрагмента; `hide()` – делает фрагмент невидимым; `show()` – отображает фрагмент. Пример реализации данных методов для двух фрагментов приведен ниже:

```

frag1 = new Fragment1();
frag2 = new Fragment2();
fTrans = getSupportFragmentManager.beginTransaction();
fTrans.add(R.id.fragmentContainer, frag1); // добавление фрагмента1
fTrans.remove(frag1); // удаление фрагмента1
fTrans.replace(R.id.fragmentContainer, frag2); // размещение фрагмента2

```

Взаимодействие фрагментов и Activity

Доступ к фрагменту из Activity осуществляется в `MainActivity.java` посредством метода `findFragmentById`:

```

// для статических фрагментов указываем id фрагмента

```

```

        Fragment        frag1        =        getSupportFragmentManager
er.findViewById(R.id.fragment1);
        TextView tv_frag1 = (TextView) frag1.getView().findViewById(R.id.tv_frag1);
        tv_frag1.setText("Hello to Fragment 1 from Activity");

        // для динамических фрагментов указываем id контейнера
        Fragment frag2 = getSupportFragmentManager.findViewById(R.id. frag-
ment_container);
        TextView tv_frag2 = (TextView) frag2.getView().findViewById(R.id.tv_frag2);
        tv_frag2.setText("Hello to Fragment 2 from Activity");

```

Доступ к Activity из фрагмента осуществляется в коде класса фрагмента с помощью метода getActivity:

```

TextView tv_activity = getActivity().findViewById(R.id.tv_activity);
tv_activity.setText("Hello from Fragment1");

```

Обработка в Activity (или в другом фрагменте) события из фрагмента выполняется посредством интерфейса в качестве прослойки для отслеживания событий. Нельзя напрямую связываться из одного фрагмента с другим!

ViewPager

Для реализации слайдера и обеспечения эффекта перелистывания страниц в Android-приложениях используется ViewPager. Каждая страница ViewPager представляет собой фрагмент. Имеется возможность формирования верхнего меню вкладок с заголовками страниц.

Алгоритм создания ViewPager следующий:

1. В activity_main.xml добавляем ViewPager, при необходимости для формирования верхнего меню вкладок с заголовками страниц дополнительно прописываем PagerTabStrip:

```

<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

```

```

<android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <android.support.v4.view.PagerTabStrip
        android:id="@+id/pagerTabStrip"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="top">
    </android.support.v4.view.PagerTabStrip>

</android.support.v4.view.ViewPager>

```

```

</RelativeLayout>

```

2. В MainActivity.java находим ViewPager и подключаем к нему PagerAdapter:

```

MainActivity extends ActionBarActivity {
    ViewPager pager;
    PagerAdapter pagerAdapter;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        pager = (ViewPager) findViewById(R.id.pager);
        pagerAdapter = new MyFragmentPagerAdapter(
            getSupportFragmentManager());
        pager.setAdapter(pagerAdapter);
    }
}

```

3. Реализуем класс MyFragmentPagerAdapter:

```

public class MyFragmentPagerAdapter extends FragmentPagerAdapter {

```

```

static final int PAGE_COUNT = 3;

public MyFragmentPagerAdapter(FragmentManager fm) {
    super(fm);
}

@Override
public Fragment getItem(int i) {
    switch (i){
        case 0: return new Fragment1();
        case 1: return new Fragment2();
        case 2: return new Fragment3();
        default: return null;
    }
}

@Override
public int getCount() {
    return PAGE_COUNT;
}

// при необходимости добавляем верхнее меню вкладок с заголовками
@Override
public CharSequence getPageTitle(int i) {
    switch (i){
        case 0: return "Frag1";
        case 1: return "Frag2";
        case 2: return "Frag3";
        default: return null;
    }
}
}

```

4. Создаем графические разметки и java-код фрагментов.
 Пример структуры проекта с ViewPager приведен на рисунке 4.3.

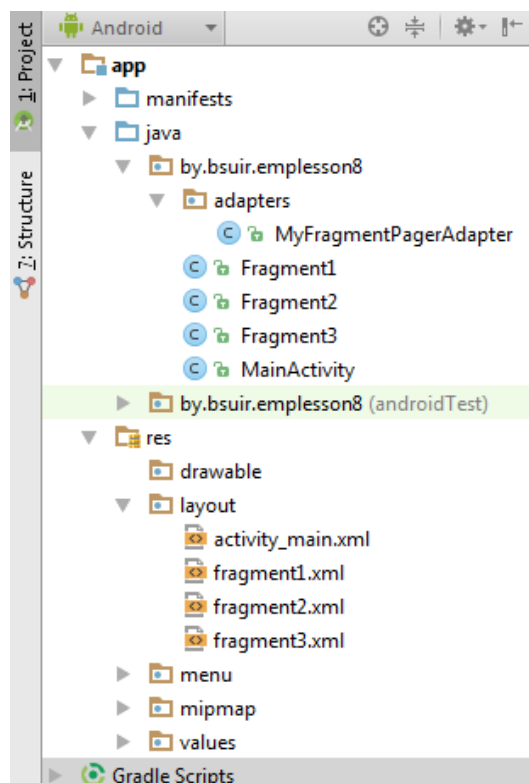


Рисунок 4.3 – Структура проекта с ViewPager

Работа с базой данных SQLite

База данных SQLite доступна на любом Android-устройстве, ее не нужно устанавливать отдельно. SQLite поддерживает типы TEXT (аналог String в Java), INTEGER (аналог long в Java) и REAL (аналог double в Java). Остальные типы следует конвертировать, прежде чем сохранять в базе данных. Для хранения изображений в базе данных указывают путь к изображениям, а сами изображения хранят в файловой системе.

Для понимания механизма работы с SQLite в Android-приложениях рассмотрим простейший пример (рисунок 4.4): создадим таблицу товаров с полями id (номер), name (наименование товара), price (цена), count (количество единиц товара в наличии). Для работы с таблицей товаров реализуем функции добавления (кнопка Add), чтения всех данных таблицы и вывода информации в лог (кнопка Read), очистки всей таблицы (кнопка Clear), ввода name, price, count в качестве новых данных уже существующего товара с номером id (кнопка Update), удаления товара по id (кнопка Delete).

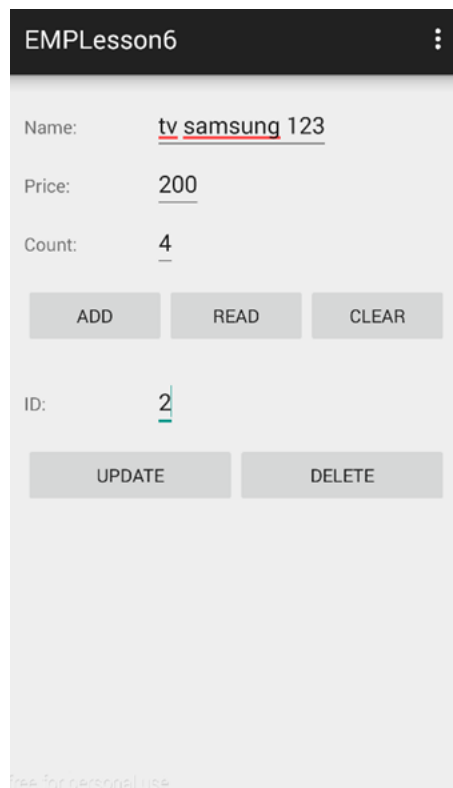


Рисунок 4.4 – Пример работы с SQLite в Android-приложениях

Работа с базой данных сводится к следующим задачам:

1. Создание и открытие базы данных, создание таблицы.

Данные задачи реализуются с помощью класса `SQLiteOpenHelper`.

2. Создание интерфейса для вставки данных.

Класс `ContentValues` используется для добавления новых строк в таблицу. Каждый объект этого класса представляет собой одну строку таблицы и выглядит как ассоциативный массив с именами столбцов и значениями, которые им соответствуют.

3. Создание интерфейса для выполнения запросов (выборки данных).

Запросы к базе данных возвращают объекты класса `Cursor`. Вместо того чтобы извлекать данные и возвращать копию значений, курсоры ссылаются на результирующий набор исходных данных. Курсоры позволяют управлять текущей позицией (строкой) в результирующем наборе данных, возвращаемом при запросе.

4. Закрытие базы данных.

Как правило выполняется в методе `onDestroy()` java-класса `Activity`.

С помощью абстрактного класса `SQLiteOpenHelper` можно создавать, открывать и обновлять базы данных. Это основной класс, с которым необходимо работать в Android-проектах.

Класс `SQLiteOpenHelper` содержит два обязательных абстрактных метода:

1) onCreate(): вызывается при первом создании базы данных;

2) onUpgrade(): вызывается при модификации базы данных (в частности, при попытке подключения к БД более новой версии, чем существующая) [1].

В приложении необходимо создать собственный класс, наследуемый от SQLiteOpenHelper. В этом классе необходимо реализовать указанные обязательные методы, описав в них логику создания и модификации используемой в приложении базы. В этом же классе принято объявлять открытые строковые константы для названия таблиц и полей создаваемой базы данных.

Для решения ранее поставленной задачи (см. рисунок 4.4) создадим класс, являющийся наследником SQLiteOpenHelper и назовем его DBHelper:

```
class DBHelper extends SQLiteOpenHelper {

    private static final String DB_TABLE = "goods";

    public DBHelper(Context context, String name, SQLiteDatabase.CursorFactory
factory, int version) {
        super(context, name, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("create table " + DB_TABLE + "("
            + "id integer primary key autoincrement,"
            + "name text,"
            + "price integer,"
            + "count integer" + ");");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // Удаляем старую таблицу и создаем новую
        db.execSQL("DROP TABLE IF IT EXISTS " + DB_TABLE);
        onCreate(db);
    }
} }
```

Реализуем вспомогательный класс DB для работы с таблицей товаров.

```
public class DB {

    private static final String DB_NAME = "mydb";
    private static final int DB_VERSION = 1;
    private static final String DB_TABLE = "goods";
    private final Context mContext;
    private DBHelper mDBHelper;
    private SQLiteDatabase mDB;

    public DB(Context ctx) {
        mContext = ctx;
    }

    // открыть подключение
    public void open() {
        mDBHelper = new DBHelper(mContext, DB_NAME, null, DB_VERSION);
        mDB = mDBHelper.getWritableDatabase();
    }

    // закрыть подключение
    public void close() {
        if (mDBHelper != null) mDBHelper.close();
    }

    // заполнить таблицу исходными данными при необходимости
    public void write() {
        int i=0;
        while (i<25) {
            i++;
            addRec("My good №" + i, i, i);
        }
    }

    // получить все данные из таблицы DB_TABLE
    public Cursor getAllData() {
```

```

        return mDB.query(DB_TABLE, null, null, null, null, null, null);
    }

    // добавить запись в DB_TABLE
    public void addRec(String name, int price, int count) {
        ContentValues cv = new ContentValues();
        cv.put("name", name);
        cv.put("price", price);
        cv.put("count", count);
        mDB.insert(DB_TABLE, null, cv);
    }

    // обновить запись в DB_TABLE
    public void update(int id, String name, int price, int count) {
        ContentValues cv = new ContentValues();
        cv.put("name", name);
        cv.put("price", price);
        cv.put("count", count);
        mDB.update(DB_TABLE, cv, "id = ?",
            new String[]{String.valueOf(id)});
    }

    // удалить запись из DB_TABLE
    public void delRec(long id) {
        mDB.delete(DB_TABLE, "id = " + id, null);
    }

    // удалить все записи из DB_TABLE
    public void delAll() {
        mDB.delete(DB_TABLE, null, null);
    }
}

```

Следует отметить, что набор методов класса DB в общем случае не является исчерпывающим и может быть изменен/дополнен при необходимости, например, методами выгрузки не всех данных, а удовлетворяющих некоторому набору условий. Для чтения данных используется метод `query()`:

Cursor query (String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String sortOrder)

В метод query() передают семь параметров, если какой-то параметр для запроса не имеет значения – указывают null:

- 1) table – имя таблицы, к которой передается запрос;
- 2) String[] columnNames – список имен возвращаемых полей (массив). При передаче null возвращаются все столбцы;
- 3) String whereClause – параметр, формирующий выражение WHERE (исключая сам оператор WHERE). Значение null возвращает все строки. Например: id = 19 and summary = ?;
- 4) String[] selectionArgs – значения аргументов фильтра. Вы можете включить ? в "whereClause". Подставляется в запрос из заданного массива;
- 5) String[] groupBy – фильтр для группировки, формирующий выражение GROUP BY (исключая сам оператор GROUP BY);
- 6) String[] having – фильтр для группировки, формирующий выражение HAVING (исключая сам оператор HAVING). Если не нужен, передается null;
- 7) String[] orderBy – параметр, формирующий выражение ORDER BY (исключая сам оператор ORDER BY). При сортировке по умолчанию передается null.

Объект Cursor, возвращаемый методом query(), обеспечивает доступ к набору записей результирующей выборки. Для обработки возвращаемых данных объект Cursor имеет набор методов для чтения каждого типа данных – getString(), getInt() и getFloat() [1-3].

Примеры поиска по имени, по стоимости:

- 1) mDB.query.query(DB_TABLE, null, " name = ?", new String[] {"Samsung Galaxy"}, null, null, null);
- 2) mDB.query(DB_TABLE, null, "count = ?", new String[] {Integer.toString(100)}, null, null, null).

Работа класса MainActivity, осуществляющего взаимодействие с базой данных, приведена далее:

```
public class MainActivity extends ActionBarActivity implements
View.OnClickListener{

    final String LOG_TAG = "myLogs";
    private Context context;
    private DB db;
    private Cursor cursor;
```

```

private int idColIndex;
private int nameColIndex;
private int priceColIndex;
private int countColIndex;

private EditText etName, etPrice, etCount, etId;
private Button btnAdd, btnRead, btnClear, btnUpdate, btnDelete;

private String name_temp;
private int id_temp, price_temp, count_temp;

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

private void initView() {
    etName = (EditText) findViewById(R.id.etName);
    etPrice = (EditText) findViewById(R.id.etPrice);
    etCount = (EditText) findViewById(R.id.etCount);
    etId = (EditText) findViewById(R.id.etId);
    btnAdd = (Button) findViewById(R.id.btnAdd);
    btnAdd.setOnClickListener(this);
    btnRead = (Button) findViewById(R.id.btnRead);
    btnRead.setOnClickListener(this);
    btnClear = (Button) findViewById(R.id.btnClear);
    btnClear.setOnClickListener(this);
    btnUpdate = (Button) findViewById(R.id.btnUpdate);
    btnUpdate.setOnClickListener(this);
    btnDelete = (Button) findViewById(R.id.btnDelete);
    btnDelete.setOnClickListener(this);
}

private void initDB() {

```

```

// открываем подключение к БД
db = new DB(this);
db.open();
//db.delAll(); // если нужно все вернуть к исходному состоянию
db.write(); // при каждом запуске дописываем данные из write()
}

```

@Override

```

public void onClick(View view) {
    switch (view.getId()){
        case R.id.btnAdd:
            name_temp = etName.getText().toString();
            price_temp = Integer.parseInt(etPrice.getText().toString());
            count_temp = Integer.parseInt(etCount.getText().toString());
            db.addRec(name_temp, price_temp, count_temp);
            break;
        case R.id.btnRead:
            // получаем курсор
            cursor = db.getAllData();
            // ставим позицию курсора на первую строку выборки
            // если в выборке нет строк, вернется false
            if (cursor.moveToFirst()) {
                // определяем номера столбцов по имени в выборке
                int idColIndex = cursor.getColumnIndex("id");
                int nameColIndex = cursor.getColumnIndex("name");
                int priceColIndex = cursor.getColumnIndex("price");
                int countColIndex = cursor.getColumnIndex("count");
                do {
                    // получаем значения по номерам столбцов и пишем в лог
                    Log.d(LOG_TAG,
                        "ID = " + cursor.getInt(idColIndex) +
                        ", name = " + cursor.getString(nameColIndex) +
                        ", price = " + cursor.getInt(priceColIndex) +
                        ", count = " + cursor.getInt(countColIndex));
                    // переход на следующую строку, а если следующей нет (те-
                    кущая - последняя), то выходим из цикла
                } while (cursor.moveToNext());
            }
    }
}

```

```

        } else {
            Log.d(LOG_TAG, "0 rows");
        }
        Log.d(LOG_TAG, "cursor.getCount()=" +
String.valueOf(cursor.getCount()));
        cursor.close();
        break;
    case R.id.btnClear:
        db.delAll();
        break;
    case R.id.btnUpdate:
        id_temp = Integer.parseInt(etId.getText().toString());
        name_temp = etName.getText().toString();
        price_temp = Integer.parseInt(etPrice.getText().toString());
        count_temp = Integer.parseInt(etCount.getText().toString());
        db.update(id_temp, name_temp, price_temp, count_temp);
        break;
    case R.id.btnDelete:
        id_temp = Integer.parseInt(etId.getText().toString());
        db.delRec(id_temp);
        break;
    }
}

protected void onDestroy() {
    super.onDestroy();
    // закрываем подключение к базе данных при выходе
    db.close();
}
}

```

Пример предоставления списком ListView данных из SQLite

Далее приведен пример реализации класса GoodsAdapter для решения задачи приложения MiniShop, ранее реализуемой посредством предоставления адаптеру набора объектов класса Good (см. рисунок 3.3). В примере реализованы паттерны проектирования CursorLoader и ViewHolder.

```
public class MainActivity extends ActionBarActivity implements OnChangeLis-
tener, View.OnClickListener, LoaderManager.LoaderCallbacks<Cursor> {
```

```
    private ListView listView;
    private DB db;
    private Cursor cursor;
    private View view_header, view_footer;
    private LayoutInflater inflater;
    private Button btnShow;
    private TextView tv_count;
    private GoodsAdapter goodsAdapter;
    private int count_checked_goods = 0;
```

```
    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        initView();
        initDB();
        createMyListView();
    }
```

```
    private void initView() {
        listView = (ListView) findViewById(R.id.listView);
    }
```

```
    private void initDB() {
        db = new DB(this);
    }
```

```
    private void createMyListView() {
        inflater = LayoutInflater.from(this);
        view_header = inflater.inflate(R.layout.header_mygoods, null);
        view_footer = inflater.inflate(R.layout.footer_mygoods, null);
        btnShow = (Button) view_footer.findViewById(R.id.btnShow);
        btnShow.setOnClickListener(this);
        tv_count = (TextView) view_footer.findViewById(R.id.tv_count);
    }
```



```

    getSupportLoaderManager().initLoader(0, null, this);
}

protected void onDestroy() {
    super.onDestroy();
    db.close();
}

@Override
public void onClick(View view) {
    Intent intent = new Intent(this, SecondActivity.class);
    startActivity(intent);
}

@Override
public void onDataChange(int id, String name, int price, int count) {
    db.update(id, name, price, count);
    getSupportLoaderManager().getLoader(0).forceLoad();
    if (count==0) {
        count_checked_goods--;
    } else {
        count_checked_goods++;
    }
    tv_count.setText("Count of goods = " + count_checked_goods + "");
}

@Override
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    return new MyCursorLoader(this, db);
}

@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
    if (goodsAdapter == null) {
        goodsAdapter = new GoodsAdapter(MainActivity.this, data, this);
        listView.addHeaderView(view_header);
        listView.addFooterView(view_footer);
    }
}

```

```

        listView.setAdapter(goodsAdapter);
    } else {
        goodsAdapter.refreshCursor(data);
    }
}

@Override
public void onLoaderReset(Loader<Cursor> loader) {
}

static class MyCursorLoader extends CursorLoader {

    DB db;

    public MyCursorLoader(Context context, DB db) {
        super(context);
        this.db = db;
    }

    @Override
    public Cursor loadInBackground() {
        Cursor cursor = db.getAllData();
        return cursor;
    }
}
}

```

```

public class GoodsAdapter extends BaseAdapter implements CompoundBut-
ton.OnCheckedChangeListener {

```

```

    private Context context;
    private LayoutInflater inflater;
    private DB db;
    private Cursor cursor;
    private int idColIndex;
    private int nameColIndex;

```

```

private int priceColIndex;
private OnChangeListener onChangeListener;

public GoodsAdapter(Context context, Cursor cursor, OnChangeListener on-
ChangeListener) {
    this.context = context;
    this.inflater = LayoutInflater.from(context);
    this.cursor = cursor;
    this.onChangeListener = onChangeListener;
    idColIndex = cursor.getColumnIndex("id");
    nameColIndex = cursor.getColumnIndex("name");
    priceColIndex = cursor.getColumnIndex("price");
}

@Override
public int getCount() {
    return cursor.getCount();
}

@Override
public Cursor getItem(int i) {
    cursor.moveToPosition(i);
    return cursor;
}

@Override
public long getItemId(int i) {
    return i;
}

@Override
public View getView(int position, View view, ViewGroup parent) {
    if (view == null) {
        view = inflater.inflate(R.layout.item_good, null);
    }
    cursor.moveToPosition(position);

```

```

ViewHolder vh = new ViewHolder();
vh.initViewHolder(view);
vh.tv_goodPrice.setText(cursor.getInt(priceColIndex)+"");
vh.tv_goodName.setText(cursor.getString(nameColIndex)+"");
if (cursor.getInt(countColIndex) == 0) {
    vh.cb_good.setChecked(false);
} else {
    vh.cb_good.setChecked(true);
}
vh.cb_good.setOnCheckedChangeListener(this);
vh.cb_good.setTag(position);
return view;
}

```

@Override

```

public void onCheckedChanged(CompoundButton compoundButton, boolean
isChecked) {
    if (compoundButton.isShown()) {
        int i = (int) compoundButton.getTag();
        cursor.moveToPosition(i);
        int id = cursor.getInt(idColIndex);
        String name = cursor.getString(nameColIndex);
        int price = cursor.getInt(priceColIndex);
        int check = 0;
        if (isChecked){
            check = 1;
        }
        onChangeListener.onDataChanged(id, name, price, check);
    }
}

```

```

public void refreshCursor(Cursor data) {
    cursor = data;
}

```

```

public class ViewHolder {

```

```

private TextView tv_goodPrice;
private TextView tv_goodName;
private CheckBox cb_good;

public ViewHolder() {
}

public void initViewHolder(View view) {
    tv_goodPrice = (TextView) view.findViewById(R.id.tv_goodPrice);
    tv_goodName = (TextView) view.findViewById(R.id.tv_goodName);
    cb_good = (CheckBox) view.findViewById(R.id.cb_good);
}
}
}

```

Практическое задание

1. Разработать приложение MyNotes, представляющее собой View Pager.
2. Поместить в View Pager четыре фрагмента: FragmentShow, FragmentAdd, FragmentDel, FragmentUpdate.
3. В View Pager добавить верхнее меню вкладок (PagerTabStrip) с заголовками Show, Add, Del, Update.
4. Во фрагменте FragmentShow реализовать кастомизированный список заметок ListView с помощью собственного адаптера.
5. В каждом пункте списка отобразить следующую информацию о заметке пользователя: номер, описание заметки.
6. Хранение, а также предоставление информации о заметках адаптеру реализовать с помощью базы данных SQLite.
7. Во фрагменте FragmentAdd реализовать функционал добавления новой заметки посредством ввода описания заметки в поле EditText и добавления информации в базу данных SQLite по нажатию кнопки Add.
8. Во фрагменте FragmentDel реализовать функционал удаления новой заметки посредством ввода ее номера в поле EditText и удаления информации из базы данных SQLite по нажатию кнопки Del.
9. Во фрагменте FragmentUpdate реализовать функционал обновления существующей заметки посредством ввода ее номера в поле EditText, ввода нового описания в поле EditText и обновления информации в базе данных SQLite по нажатию кнопки Update.

10. Предусмотреть обработку исключительной ситуации отсутствия заметки по указанному номеру посредством вывода пользователю всплывающего сообщения соответствующего содержания.

11. Продемонстрировать работу приложения MyNotes на эмуляторе или реальном устройстве.

12. Дополнительное задание, предполагающее самостоятельное углубленное освоение материала: реализовать приложение MiniShop с помощью фрагментов и базы данных SQLite. Предусмотреть различное расположение фрагментов в портретной и альбомной ориентациях (см. рисунок 4.1).

Содержание отчета

1. Скриншоты графических представлений фрагментов FragmentShow, FragmentAdd, FragmentDel, FragmentUpdate в Android Studio, демонстрирующие логику работы приложения MyNotes.

2. Код xml-файлов графических представлений, используемых в приложении MyNotes.

3. Код java-файлов приложения MyNotes, включая классы для работы с базой данных, Activity, фрагменты, адаптеры.

Контрольные вопросы

1. Что представляет собой фрагмент? Для чего нужны фрагменты?

2. Как статически добавить фрагмент в Activity?

3. Как динамически добавить фрагмент в Activity?

4. Какие динамические операции возможны с фрагментами?

5. Что такое ViewPager?

6. Как создать ViewPager?

7. Для чего используются PagerTitleStrip и PagerTabStrip?

8. Как называется базовый класс для работы с базой данных SQLite в Android?

9. Какие методы обязательны для переопределения при работе с базой данных SQLite?

10. Для чего используется класс ContentValues?

11. Для чего используется класс Cursor?

12. Как реализуются методы insert, query, delete, update для вставки, чтения, удаления и добавления записи в SQLite.

Литература

1. Google Android это несложно [Электронный ресурс]. – 2011. – Режим доступа : <http://startandroid.ru>. – Дата доступа : 10.03.2017.
2. FANDROID.info [Электронный ресурс]. – 2014. – Режим доступа : <http://www.fandroid.info/>. – Дата доступа : 10.03.2017.
3. Освой программирование играючи. Сайт Александра Климова [Электронный ресурс]. – 2000. – Режим доступа : <http://developer.alexanderklimov.ru/>. – Дата доступа : 10.03.2017.
4. Android Developers [Электронный ресурс]. – 2008. – Режим доступа : <https://developer.android.com/index.html>. – Дата доступа : 10.03.2017.
5. Голощапов, А. Л. Google Android: программирование для мобильных устройств / А. Л. Голощапов. – СПб. : БХВ – Петербург, 2011. – 448 с.
6. Ed Burnette Hello, Android. The Pragmatic Bookshelf. – Dallas, Texas, 2010. – 302 p.
7. Дэрси, Л. Android за 24 часа. Программирование приложений под операционную систему Google / Л. Дэрси, Ш. Кондер. – М. : Рид Групп, 2011. – 464 с.

Учебное издание

Меженная Марина Михайловна
Быков Антон Алексеевич
Каландаров Артур Игоревич
Гордейчук Татьяна Валерьевна

**ЭРГНОМИКА МОБИЛЬНЫХ ПРИЛОЖЕНИЙ.
ЛАБОРАТОРНЫЙ ПРАКТИКУМ**

ПОСОБИЕ

Редактор М. А. Зайцева
Корректор
Компьютерная правка, оригинал-макет

Подписано в печать. Формат 60х84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечано на ризографе. Усл. печ. л. . Уч.-изд.л. . Тираж 50 экз. Заказ 143.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
ЛП №02330/264 от 14.04.2014.
220013, Минск, П. Бровки, 6