

**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ**

**РОССИЙСКОЙ ФЕДЕРАЦИИ**

Брянский государственный технический университет

**Утверждаю**

**Ректор университета**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_О.Н.Федонин**

**«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_2017 г.**

**Программирование МОБИЛЬНЫХ СИСТЕМ**

**ХРАНЕНИЕ ПРОСТЫХ ДАННЫХ в ANDROID. БАЗЫ ДАННЫХ В ANDROID**

**Методические указания**

**к выполнению лабораторной работы №4**

**для студентов очной формы обучения по направлениям подготовки 02.03.03 – «Математическое обеспечение и администрирование информационных систем», 09.03.01 – «Информатика и вычислительная техника» и 09.03.04 – «Программная инженерия»**

**Брянск 2017**

УКД 004.43

Программирование мобильных систем. Знакомство с базами данных и источниками данных в ОС Android. [Текст] + [Электронный ресурс]: Методические указания к выполнению лабораторной работы №5 для студентов очной формы обучения по направлениям подготовки 02.03.03 – «Математическое обеспечение и администрирование информационных систем», 09.03.01 – «Информатика и вычислительная техника» и 09.03.04 – «Программная инженерия». – Брянск: БГТУ, 2017. – 18 с.

Разработал:

Д.Н.Панус

ст.преп.

Рекомендовано кафедрой «Информатика и программное обеспечение» БГТУ (протокол №2 от 16.09.2016)

Научный редактор Д.А.Коростелев

Редактор издательства Л.И.Афонина

Компьютерный набор Д.Н.Панус

Темплан 2017 г., п.273

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подписано в печать 23.09.17. Формат 60х84 1/16 Бумага офсетная. Офсетная печать. Усл.печ.л. 1,8. Уч.-изд.л. 1,8 Тираж 1 экз. Заказ Бесплатно.

Издательство Брянского государственного технического университета

241035, Брянск, бульвар 50-летия Октября, 7, БГТУ. 58-82-49.

Лаборатория оперативной полиграфии БГТУ, ул. Институтская, 16.

# Цель работы

Цель работы – освоение средств ОС для хранения, обработки и предоставление информации приложению на устройстве.

Продолжительность работы - 3 часа.

**Порядок выполнения работы**

1. Ознакомиться с теоретическим материалом для лабораторной работы. Разобраться с теорией хранения пользовательской и системной информации приложениями в ОС Android.
2. Произвести проектирование и реализацию интерфейса приложения.
3. Реализовать механизм хранение простых значений через класс SharedPreferences.
4. Разобраться с уровнями доступа к механизму общих настроек.
5. Ознакомиться с особенностями взаимодействия с SQLite и его применения в ОС Android.
6. Реализовать в приложении требования к лабораторной работе.
7. Произвести сборку готового приложения в виде \*.apk файла приложения.
8. Продемонстрировать полученный результат преподавателю.

**Работа с файловой системой Android. Чтение и сохранение файлов.**

Работа с настройками уровня *activity* и приложения позволяет сохранить небольшие данные отдельных типов (*string, int*), но для работы с большими массивами данных, такими как графически файлы, файлы мультимедиа и т.д., нам придется обращаться к файловой системе.

ОС Android построена на основе Linux. Этот факт находит свое отражение в работе с файлами. Все названия файлов и каталогов являются регистрозависимыми, то есть "data" это не то же самое, что и "Data".

Приложение Android сохраняет свои данные в каталоге */data/data/<название\_пакета>/* и, как правило, относительно этого каталога будет идти работа.

Для работы с файлами абстрактный класс *android.content.Context* определяет ряд методов:

* **deleteFile(String name)**: удаляет определенный файл
* **fileList()**: получает все файлы, которые содержатся в подкаталоге /files в каталоге приложения
* **getCacheDir()**: получает ссылку на подкаталог cache в каталоге приложения
* **getDir(String dirName, int mode)**: получает ссылку на подкаталог в каталоге приложения, если такого подкаталога нет, то он создается
* **getExternalCacheDir()**: получает ссылку на папку /cache внешней файловой системы устройства
* **getExternalFilesDir()**: получает ссылку на каталог /files внешней файловой системы устройства
* **getFileStreamPath(String filename)**: возвращает абсолютный путь к файлу в файловой системе
* **openFileInput(String filename)**: открывает файл для чтения
* **openFileOutput (String name, int mode)**: открывает файл для записи

Все файлы, которые создаются и редактируются в приложении, как правило, хранятся в подкаталоге */files* в каталоге приложения.

Для непосредственного чтения и записи файлов применяются также стандартные классы *Java* из пакета *java.io*.

Итак, применим функционал чтения-записи файлов в приложении. Пусть у нас будет следующая разметка layout:

|  |
| --- |
| <LinearLayout      android:layout\_width="match\_parent"      android:layout\_height="match\_parent"      android:orientation="vertical">      <EditText          android:id="@+id/save\_text"          android:layout\_width="match\_parent"          android:layout\_height="0dp"          android:gravity="start"          android:layout\_weight="4"/>      <Button          android:layout\_width="wrap\_content"          android:layout\_height="0dp"          android:layout\_weight="1"          android:layout\_margin="16dp"          android:layout\_gravity="center"          android:onClick="saveText"          android:text="Сохранить"/>       <TextView          android:layout\_marginTop="80dp"          android:id="@+id/open\_text"          android:layout\_width="match\_parent"          android:layout\_height="0dp"          android:gravity="start"          android:layout\_weight="4"/>      <Button          android:layout\_width="wrap\_content"          android:layout\_height="0dp"          android:layout\_margin="16dp"          android:layout\_weight="1"          android:layout\_gravity="center"          android:onClick="openText"          android:text="Открыть"/>  </LinearLayout> |

Поле *EditText* предназначено для ввода текста, а *TextView* - для вывода ранее сохраненного текста. Для сохранения и восстановления текста добавлены две кнопки.

Теперь в коде *Activity* пропишем обработчики кнопок с сохранением и чтением файла:

|  |
| --- |
| package com.example.app;  import android.support.v7.app.AppCompatActivity;  import android.os.Bundle;  import android.view.View;  import android.widget.EditText;  import android.widget.TextView;  import android.widget.Toast;    import java.io.FileInputStream;  import java.io.FileOutputStream;  import java.io.IOException;    public class MainActivity extends AppCompatActivity {      private final static String FILE\_NAME = "content.txt";      @Override      protected void onCreate(Bundle savedInstanceState) {          super.onCreate(savedInstanceState);          setContentView(R.layout.activity\_main);      }      // сохранение файла      public void saveText(View view){          FileOutputStream fos = null;          try {              EditText textBox = (EditText) findViewById(R.id.save\_text);              String text = textBox.getText().toString();                fos = openFileOutput(FILE\_NAME, MODE\_PRIVATE);              fos.write(text.getBytes());              Toast.makeText(this, "Файл сохранен", Toast.LENGTH\_SHORT).show();          }          catch(IOException ex) {              Toast.makeText(this, ex.getMessage(), Toast.LENGTH\_SHORT).show();          }          finally{              try{                  if(fos!=null)                      fos.close();              }              catch(IOException ex){                    Toast.makeText(this, ex.getMessage(), Toast.LENGTH\_SHORT).show();              }          }      }      // открытие файла      public void openText(View view){          FileInputStream fin = null;          TextView textView = (TextView) findViewById(R.id.open\_text);          try {              fin = openFileInput(FILE\_NAME);              byte[] bytes = new byte[fin.available()];              fin.read(bytes);              String text = new String (bytes);              textView.setText(text);          }          catch(IOException ex) {              Toast.makeText(this, ex.getMessage(), Toast.LENGTH\_SHORT).show();          }          finally{              try{                  if(fin!=null)                      fin.close();              }              catch(IOException ex){                  Toast.makeText(this, ex.getMessage(), Toast.LENGTH\_SHORT).show();              }          }      }  } |

При нажатии на кнопку сохранения будет создаваться поток вывода *FileOutputStream fos = openFileOutput(FILE\_NAME, MODE\_PRIVATE)*

В данном случае введенный текст будет сохраняться в файл "content.txt". При этом будет использоваться режим *MODE\_PRIVATE.*

Система позволяет создавать файлы с двумя разными режимами:

* **MODE\_PRIVATE:** файлы могут быть доступны только владельцу приложения (режим по умолчанию)
* **MODE\_APPEND:** данные могут быть добавлены в конец файла

Поэтому в данном случае если файл "content.txt" уже существует, то он будет перезаписан. Если же нам надо было дописать файл, тогда надо было бы использовать режим *MODE\_APPEND*:

|  |
| --- |
| FileOutputStream fos = openFileOutput(FILE\_NAME, MODE\_APPEND); |

Для чтения файла применяется поток ввода *FileInputStream*:

|  |
| --- |
| FileInputStream fin = openFileInput(FILE\_NAME); |

**Сохранение простых данных приложения. Класс SharedPreferences.**

Есть два «легковесных» метода сохранения простых программных данных в приложениях для Android – Общие настройки и пара обработчиков событий, предназначенных для сохранения информации об экземпляре Активности. Оба механизма используют принцип пар «ключ-значение», чтобы хранить простые, примитивные значения.

Класс *SharedPreferences* позволяет создавать в приложении именованные ассоциативные массивы типа «ключ-значение», которые могут быть использованы различными компонентами приложения.

Общие настройки поддерживают базовые *типы Boolean, string, float, long, integer*, что делает их идеальным средством для быстрого сохранения значений по умолчанию, переменных экземпляра класса, текущего состояния UI и пользовательских настроек. Они чаще всего используются для обеспечения постоянства данных между пользовательскими сессиями и доступа к ним компонентов приложения.

Активности также предоставляют обработчик *onSaveInstanceState* – специально для сохранения состояния пользовательского интерфейса, когда работа Активности может быть завершена из-за нехватки ресурсов.

Этот обработчик функционирует по тому же принципу, что и общие настройки, и предлагает параметр *Bundle* – ассоциативный массив вида «ключ-значение». *Bundle* – параметр, который передается в методы *onCreate* и *onRestoreInstanceState*. Bundle, содержащий состояние UI, должен быть использован для записи значений, необходимых Активности, чтобы вывести на экран тот самый пользовательский интерфейс, который отображался до непредвиденного закрытия.

Класс *android.content.SharedPreferences* предоставляет ряд методов для управления настройками:

* **contains(String key):** возвращает true, если в настройках сохранено значение с ключом key;
* **getAll():** возвращает все сохраненные в настройках значения;
* **getBoolean (String key, boolean defValue):** возвращает из настроек значение типа Boolean, которое имеет ключ key. Если элемента с таким ключом не окажется, то возвращается значение defValue, передаваемое вторым параметром;
* **getFloat(String key, float defValue):** возвращает значение типа float с ключом key. Если элемента с таким ключом не окажется, то возвращается значение defValue;
* **getInt(String key, int defValue):** возвращает значение типа int с ключом key;
* **getLong(String key, long defValue):** возвращает значение типа long с ключом key;
* **getString(String key, String defValue):** возвращает строковое значение с ключом key;
* **getStringSet(String key, Set<String> defValues):** возвращает массив строк с ключом key;
* **edit():** возвращает объект SharedPreferences.Editor, который используется для редактирования настроек.

**Создание и сохранение настроек.**

Чтобы создать или изменить общие настройки, необходимо вызвать метод *getSharedPreferences*в конте**к**сте приложения, передавая имя общих настроек, которые вы хотите изменить. Общие настройки доступны для компонентов приложения, но не для других приложений.

Для управления настройками используется объект класса *SharedPreferences.Editor*, возвращаемый метод *edit()****.*** Он определяет следующие методы:

* **clear():** удаляет все настройки
* **remove(String key):** удаляет из настроек значение с ключом key
* **putBoolean(String key, boolean value**): добавляет в настройки значение типа boolean с ключом key
* **putFloat(String key, float value):** добавляет в настройки значение типа float с ключом key
* **putInt(String key, int value):** добавляет в настройки значение int с ключом key
* **putLong(String key, long value):** добавляет в настройки значение типа long с ключом key
* **putString(String key, String value):** добавляет в настройки строку с ключом key
* **putStringSet(String key, Set<String> values):** добавляет в настройки строковый массив
* **commit():** подтверждает все изменения в настройках
* **apply():** также, как и метод commit(), подтверждает все изменения в настройках, однако измененный объект SharedPreferences вначале сохраняется во временной памяти, и лишь затем в результате асинхронной операции записывается на мобильное устройство.

|  |
| --- |
| //Получите доступ к объекту Editor, чтобы изменить общие настройки.  SharedPreferences.Editor editor = mySharedPreferences.edit();  //Задайте новые базовые типы в объекте общих настроек.  editor.putBoolean(“isTrue”, true);  editor.putFloat(“lastFloat”, lf);  editor.putInt(“wholeNumber”, 2);  editor.putLong(“aNumber”, 3l);  editor.putString(“textEntryValue”, “Not Empty”);  //Сохраните изменения.  editor.commit(); |

Нередко возникает задача автоматически сохранять вводимые данные при выходе пользователя из *activity*. Для этого можно переопределить метод *onPause*:

|  |
| --- |
| package com.example.app;  import android.content.SharedPreferences;  import android.support.v7.app.AppCompatActivity;  import android.os.Bundle;  import android.view.View;  import android.widget.EditText;    public class MainActivity extends AppCompatActivity {      private static final String PREFS\_FILE = "Account";      private static final String PREF\_NAME = "Name";      EditText nameBox;      SharedPreferences settings;      SharedPreferences.Editor prefEditor;      @Override      protected void onCreate(Bundle savedInstanceState) {          super.onCreate(savedInstanceState);          setContentView(R.layout.activity\_main);          nameBox = (EditText) findViewById(R.id.nameBox);          settings = getSharedPreferences(PREFS\_FILE, MODE\_PRIVATE);          // получаем настройки          String name = settings.getString(PREF\_NAME,"");          nameBox.setText(name);      }      @Override      protected void onPause(){          super.onPause();            EditText nameBox = (EditText) findViewById(R.id.nameBox);          String name = nameBox.getText().toString();          // сохраняем в настройках          prefEditor = settings.edit();          prefEditor.putString(PREF\_NAME, nameBox.getText().toString());          prefEditor.apply();      }        public void saveName(View view) {      }        public void getName(View view) {      }  } |

**Приватные настройки**

Кроме общих настроек каждая activity может использовать приватные, к которым доступ из других activity будет невозможен. Для получения настроек уровня activity используется метод *getPreferences(MODE\_PRIVATE)*:

|  |
| --- |
| import android.content.SharedPreferences;  //........................  SharedPreferences settings = getPreferences(MODE\_PRIVATE); |

То есть в отличие от общих настроек здесь не используется название группы настроек в качестве первого параметра, как в методе *getSharedPreferences().* Однако вся остальная работа по добавлению, получению и изменению настроек будет аналогична работает с общими настройками.

**Знакомство с базами данных**

Android обеспечивает долгосрочное хранение структурированной информации с помощью комбинации баз данных SQLite и источников данных.

SQLite позволяет создавать для приложения независимые реляционные базы данных и использовать их для хранения сложных структурированных данных и управления ими.

Источники данных предоставляют интерфейс для публикации и потребления данных, основанных на адресной модели URI.

**Курсоры и класс ContentValues**

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.

Запросы к базе данных возвращают объекты класса Cursor, которые ссылаются на результирующий набор исходных данных и позволяют управлять текущей позицией (строкой) в запросе.

**Знакомство с Адаптерами**

Адаптеры используются для привязки данных к Группам представлений, которые наследуют класс *AdapterView*. Можно создавать собственные классы Адаптеров и конструировать новые элементы управления на основе *AdapterView*.

Поскольку Адаптеры отвечают и за доставку данных, и за создание Представлений для их отображения, они могут сильно повлиять на внешний вид и функциональность элемента, к которому привязаны.

Существует два наиболее полезных и универсальных стандартных Адаптера:

* **ArrayAdapter**. Использует механизм обобщенных типов для привязки *AdapterView* к массиву объектов определенного класса. По умолчанию этот Адаптер задействует метод toString для каждого объекта в массиве, чтобы создать и наполнить данными элементами *TextView*.
* **SimpleCursorAdapter**. Прикрепляет Представление, заданное внутри разметки, к столбцам Курсора, ставшего результатом запроса к источнику данных. Сначала нужно описать разметку в формате XML, чтобы, загрузив ее, вывести на экран содержимое каждого дочернего Представления. Затем нужно привязать каждый столбец Курсора к Представлению из этой разметки. Адаптер создаст Представления для каждой записи Курсора и наполнит их данными из соответствующего столбца.

**Класс SQLiteOpenHelper**

Библиотека Android содержит абстрактный класс *SQLiteOpenHelper*, с помощью которого можно создавать, открывать и обновлять базы данных. Это основной класс, с которым придётся сталкиваться в проектах. При реализации этого вспомогательного класса обычно скрывается логика, на основе которой принимается решение о создании или обновлении базы данных перед ее открытием. Класс *SQLiteOpenHelper* содержит два обязательных абстрактных метода:

* **onCreate()** — вызывается при первом создании базы данных
* **onUpgrade()** — вызывается при модификации базы данных

Также используются другие методы класса:

* **onDowngrade(SQLiteDatabase, int, int)**
* **onOpen(SQLiteDatabase)**
* **getReadableDatabase()**
* **getWritableDatabase()**

В приложении необходимо создать собственный класс, наследуемый от *SQLiteOpenHelper*. В этом классе необходимо реализовать указанные обязательные методы, описав в них логику создания и модификации вашей базы.

В этом же классе принято объявлять открытые строковые константы для названия таблиц и полей создаваемой базы данных, которые клиенты могут использовать для определения столбцов при выполнении запросов к базе данных. Например, так можно объявить константы для таблицы *CONTACT*:

|  |
| --- |
| public static final String TABLE\_NAME = "CONTACT";  public static final String NAME = "FIRST\_NAME";  public static final String PHONE = "PHONE"; |

Лучше сразу давать понятные имена, указывающие на работу с таблицей. Если имя переменной **TABLE\_NAME** ещё можно оставить, то для других лучше использовать более говорящие имена, например, **KEY\_NAME** и **KEY\_PHONE** или **COLUMN\_NAME**, **COLUMN\_PHONE**.

Ваш класс, расширяющий *SQLiteOpenHelper*, также неявно наследует интерфейс *BaseColumns*, в котором определена строковая константа **\_ID**, представляющая имя поля для идентификаторов записей. В создаваемых таблицах базы данных поле \_ID должно иметь тип **INTEGER PRIMARY KEY AUTOINCREMENT**. Описатель **AUTOINCREMENT** является необязательным. Часто в других примерах идентификатор создаётся вручную, смотрите как вам удобнее. Только всегда называйте его именно **\_id**. Такое название используется в Android для работы с курсорами и поэтому придерживайтесь данного правила.

В методе *onCreate()* необходимо реализовать логику создания таблиц и при необходимости заполнить их начальными данными при помощи SQL-команды, например:

|  |
| --- |
| @Override  public void onCreate(SQLiteDatabase db)  {  db.execSQL("CREATE TABLE + TABLE\_NAME  (\_id INTEGER PRIMARY KEY AUTOINCREMENT,  COLUMN\_NAME TEXT,  COLUMN\_PHONE TEXT);");  } |

Здесь показана условная команда, в реальном коде вам нужно учитывать пробелы между константами, чтобы получилась правильная строка SQL-команды.

Также надо указать номер версии базы данных, начиная со значения 1. По этому номеру класс-обёртка будет понимать, что структуру базы данных следует обновить.

|  |
| --- |
| private static final int DATABASE\_VERSION = 1; |

В методе *onUpgrade()* можно, например, реализовать запрос в базу данных на уничтожение таблицы (DROP TABLE), после чего вновь вызвать метод *onCreate()* для создания версии таблицы с обновленной структурой, например, так:

|  |
| --- |
| @Override  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)  {  db.execSQL("DROP TABLE IF EXISTS + TABLE\_NAME);  onCreate(db);  } |

В параметрах метода используются номера версий базы данных - старая и новая. О номере версии говорилось выше.

Обновляется структура базы данных следующим образом. Сначала меняем порядковый номер.

|  |
| --- |
| private static final int DATABASE\_VERSION = 2; |

При первой установке приложения базы данных ещё не существует. Тут проверять пока нечего. При установке новой версии приложения система проверит номер версии базы данных. Если он больше, чем было, то вызовется метод *onUpgrade()*. Если наоборот, то вызовется метод *onDowngrade()* (обычно так происходит редко). Просто ставим условие на проверку.

|  |
| --- |
| @Override  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {  if (oldVersion == 1) {  // Код для работы с базой данных с версией 1  }  if (oldVersion < 3) {  // Код для работы с базой данных с версией 1 или 2  }  } |

Допустим, в таблицу нужно добавить новую колонку в дополнение к существующим колонкам NAME, DESCRIPTION и IMAGE\_RESOURCE\_ID. Для изменения таблицы используется выражение SQL с ключевым словом **ALTER**

|  |
| --- |
| ALTER TABLE DATABASE\_TABLE ADD COLUMN AGE INTEGER |

Для замены имени таблицы используется выражение (меняем имя таблицы DOG на CAT):

|  |
| --- |
| ALTER TABLE DOG RENAME TO CAT |

Для удаления таблицы:

|  |
| --- |
| DROP TABLE DOG |

Соответственно, нужно вызвать метод *SQLiteDatabase.execSQL()*, передав ему нужную команду SQL.

**Класс SQLiteDatabase**

Для управления базой данных SQLite существует класс *SQLiteDatabase*. В классе *SQLiteDatabase* определены методы *query()*, *insert()*, *delete()* и *update()* для чтения, добавления, удаления, изменения данных. Кроме того, метод *execSQL()* позволяет выполнять любой допустимый код на языке SQL применимо к таблицам базы данных, если вы хотите провести эти (или любые другие) операции вручную.

Каждый раз, когда вы редактируете очередное значение из базы данных, нужно вызывать метод *refreshQuery()* для всех курсоров, которые имеют отношение к редактируемой таблице.

Для составления запроса используются два метода: *rawQuery()* и *query()*, а также класс *SQLiteQueryBuilder*.

**Метод openOrCreateDatabase**

Вы можете открывать и создавать базы данных без помощи класса *SQLiteOpenHelper*, используя метод *openOrCreateDatabase()*, принадлежащий объекту *Context* вашего приложения. Получите доступ к базе данных в два шага. Сначала вызовите метод *openOrCreateDatabase()*, чтобы создать новую базу данных. Затем из полученного экземпляра базы данных вызовите *execSQL()*, чтобы выполнять команды на языке SQL, с помощью которых будут созданы таблицы и установлены отношения между ними.

|  |
| --- |
| private static final String DATABASE\_NAME = "myDatabase.db";  private static final String DATABASE\_TABLE = "mainTable";  private static final String DATABASE\_CREATE =  "create table " + DATABASE\_TABLE + " ( \_id integer primary key autoincrement," +  "column\_one text not null);";  SQLiteDatabase myDatabase;  private void createDatabase() {  myDatabase = openOrCreateDatabase(DATABASE\_NAME, Context.MODE\_PRIVATE,  null);  myDatabase.execSQL(DATABASE\_CREATE);  } |

**Задание на лабораторную работу**

Необходимо доработать в своем Android приложении, способность сохранять результаты своей работы средствами базовых API и подходов: общие настройки, файлы и подключаемые ресурсы, со следующими требованиями:

1. Приложение должно сохранять состояние своих данных при обработке событий жизненного цикла, способных привести к их потери. (Должны сохраняться любые не статичные данные, отображаемые или вводимые пользователем). При загрузке все данные должны быть восстановлены и отображены на экране по своим местам.
2. При наличии в оригинальном приложении экрана настроек, необходимо реализовать его в своем приложении (PreferencesScreen).
3. Добавить в качестве ресурсов текстовый файл (\*.csv, \*.json, \*.xml ). Данные из этого файла должны стать начальным состоянием базы данных приложения и отображены на соответствующих экранах.
4. При наличии в приложении, в любой его части, указания на импорт экспорт базы данных это необходимо реализовать.
5. В приложении реализовать SQLite базу данных со следующими требованиями: БД должна соответствовать оригинальному приложению, таблицы должны иметь все необходимые связи между собой. Количество таблиц, которое ожидается не менее 3.
6. Для отображения записей из БД использовать адаптеры. В адаптере реализовать элементы взаимодействия с пользователем, если такой функционал там имеется.
7. При отображении связанных данных обязательно должно использоваться естественное представление этих данных, а не индекс записи из смежной таблицы.
8. Настоятельно рекомендуется для работы с БД использовать ORM библиотеки (например, dbflow https://github.com/Raizlabs/DBFlow). К снижению оценки отказ от использования не приведет, а вот к увеличению времени на исполнение - да.

**Контрольные вопросы**

1. Механизм общих настроек. Зачем нужен и из чего состоит?
2. Сохранение состояния Activity
3. Работа с файлами в Android. Приватная зона хранения файлов в Android.
4. Класс SharedPreferences назначение и особенности.
5. Создание, хранение и получение настроек.
6. PreferencesScreen: стандартные возможности предоставляемые через разметку. PreferencesActivity.
7. Отслеживание изменений в общих настройках
8. Что такое адаптеры и зачем они нужны
9. Основные виды адаптеров их отличительные свойства.
10. Основные способы хранения данных в OS android.
11. Что такое источник данных.
12. Какие основные достоинства применения SQLite в мобильных устройствах. Местоположение файлов базы данных.
13. Ограничения SQLite.
14. Принципы работы с Базами данных в OS android. Класс Cursor.
15. Класс SQLiteOpenHelper.
16. Особенность выполнения запросов к базе данных при помощи методов query insert update и delete
17. Класс ContentValues. Его назначение.
18. Связь жизненного цикла курсора и activity. Как делается и зачем необходимо?

# Список рекомендуемой литературы

# Майер, Р. Android 4. Программирование приложений для планшетных компьютеров и смартфонов. / Пер. с англ. – М.: Эксмо, 2013. – 816 С.

# Коматинени, С. Android 4 для профессионалов. Создание приложений для планшетных компьютеров и смартфонов./ Коматинени С., Маклин Д. / Пер. с англ. – М.: Вильямс, 2012. – 880 С.