Хранение данных

Способы хранения информации

- **Preferences** – настройки уровня приложения
- **Файлы** - внутренние и внешние (на SD карте)
- **SQLite** - база данных, таблицы

Preferences

Значения сохраняются в виде пары:
имя - значение.

Также, как и например extras в Intent.

Сохранение. Пример

```
public class MainActivity extends Activity

  EditText etText;
  Button btnSave, btnLoad;

  SharedPreferences sPref;

  final String SAVED_TEXT = "saved_text";
```

```
void saveText() {
  sPref = getPreferences(MODE_PRIVATE);
  Editor ed = sPref.edit();
  ed.putString(SAVED_TEXT, etText.getText().toString());
  ed.commit();
}
```

Загрузка. Пример

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

Файлы

Работа с настройками (Preferences) 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.

Android и Linux

ОС Android построена на основе Linux. В путях к файлам в качестве разграничителя в Linux использует прямой слеш "/", а не обратный слеш "\" (как в Windows). А 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.

Файлы каталога приложения

Работа с файлами каталога приложения в Android не сильно отличается от таковой в Java. По умолчанию такие файлы доступны только самому приложения.

Функции Android для работы с файлами:

- **openFileInput**(String filename): открывает файл для чтения

- **openFileOutput** (String name, int mode): открывает файл для записи

Функции работы с файлами

- **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

Функции работы с файлами

- **getExternalCacheDir**(): получает ссылку на папку */cache* внешней файловой системы
- **getExternalFilesDir**(): получает ссылку на каталог */files* внешней файловой системы
- **getFileStreamPath**(String filename): возвращает абсолютный путь к файлу в файловой системе

Запись в файл

```
void writeFile() {
  try {
    // отрываем поток для записи
    BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(
        openFileOutput(FILENAME, MODE_PRIVATE)));
    // пишем данные
    bw.write("Содержимое файла");
    // закрываем поток
    bw.close();
    Log.d(LOG_TAG, "Файл записан");
  } catch (FileNotFoundException e) {
    e.printStackTrace();
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Читаем из файла

```
void readFile() {
  try {
    // открываем поток для чтения
    BufferedReader br = new BufferedReader(new InputStreamReader(
        openFileInput(FILENAME)));
    String str = "";
    // читаем содержимое
    while ((str = br.readLine()) != null) {
      Log.d(LOG_TAG, str);
    }
  } catch (FileNotFoundException e) {
    e.printStackTrace();
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Файлы во внешнем хранилище

Кроме каталога приложения мы можем работать с файлами из внешнего хранилища. Это позволит другим программам открывать данные файлы и при необходимости изменять. Весь механизм работы с файлами будет таким же, как и при работе с каталогом приложения. Ключевым отличием здесь будет получение и использование пути к внешнему хранилищу через Environment:

Environment.getExternalStorageDirectory()

Файлы во внешнем хранилище

Поскольку для чтения/записи во внешнее хранилище необходимы разрешения, то перед операциями сохранения и записи файла необходимо проверить наличие разрешений. Для этого определен метод **checkPermissions**(). При установке разрешений срабатывает метод **onRequestPermissionsResult**(), в котором в случае удачной установки разрешений для переменной **permissionGranted** задается значение true.

Разрешения (Permissions)

Чтобы использовать внешнее хранилище, также надо установить разрешения в файле манифеста AndroidManifest.xml:

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

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

Запись файла на SD

```
// получаем путь к SD
File sdPath = Environment.getExternalStorageDirectory();
// добавляем свой каталог к пути
sdPath = new File(sdPath.getAbsolutePath() + "/" + DIR_SD);
// создаем каталог
sdPath.mkdirs();
// формируем объект File, который содержит путь к файлу
File sdFile = new File(sdPath, FILENAME_SD);
try {
  // открываем поток для записи
  BufferedWriter bw = new BufferedWriter(new FileWriter(sdFile));
  // пишем данные
  bw.write("Содержимое файла на SD");
  // закрываем поток
  bw.close();
  Log.d(LOG_TAG, "Файл записан на SD: " + sdFile.getAbsolutePath());
} catch (IOException e) {
  e.printStackTrace();
}
```

Чтение файла с SD

```
// получаем путь к SD
File sdPath = Environment.getExternalStorageDirectory();
// добавляем свой каталог к пути
sdPath = new File(sdPath.getAbsolutePath() + "/" + DIR_SD);
// формируем объект File, который содержит путь к файлу
File sdFile = new File(sdPath, FILENAME_SD);
try {
  // открываем поток для чтения
  BufferedReader br = new BufferedReader(new FileReader(sdFile));
  String str = "";
  // читаем содержимое
  while ((str = br.readLine()) != null) {
    Log.d(LOG_TAG, str);
  }
} catch (FileNotFoundException e) {
  e.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
}
```

Проверка SD

```
// проверяем доступность SD
if (!Environment.getExternalStorageState().equals(
    Environment.MEDIA_MOUNTED)) {
  Log.d(LOG_TAG, "SD-карта не доступна"
  return;
}
```

Структурированные данные

Настройки дают самый простой способ хранения небольшой информации.
Файлы значительно расширяют наши возможности по работе с данными: можно работать с картинками, видео и крупными текстовыми массивами и т.д.

Однако и настройки, и файлы достаточно ограничены и для хранения большого количества **структурированных** данных неудобны.

База данных SQLite

В Android имеется встроенная поддержка одной из распространенных систем управления базами данных - SQLite. Это база данных с таблицами и запросами - все как в обычных БД. Почему именно SQLite? Преимущества SQLite:

- **Минимальные затраты ресурсов**
- **Оптимизация для одного пользователя**
- **Надежность и быстрота**

SQLite: использование

ОС Android по умолчанию уже содержит ряд встроенных баз SQLite, которые используются стандартными программами - для списка контактов, для хранения фотографий с камеры, музыкальных альбомов и т.д.

Наша база данных будет храниться в каталоге приложения по пути:
DATA/data/[Название_приложения]/databases/[Название_файла_базы_данных]

SQLite: структура файлов

Каждая база данных состоит из двух файлов.
1) Первый файл — файл базы данных (соответствует имени базы данных). Это основной файл базы данных SQLite; в нем хранятся все данные.

2) Второй файл — файл журнала. Его имя состоит из имени базы данных и суффикса “-journal” — например, “mybd-journal”. В файле журнала хранится информация обо всех изменениях, внесенных в базу данных.

SQLite: структура файлов

Задачи работы с СУБД

- Создание и открытие базы данных
- Создание таблицы
- Создание интерфейса для вставки данных (insert)
- Создание интерфейса для выполнения запросов (выборка данных)
- Закрытие базы данных

Создание и открытие базы

Для создания или открытия новой базы данных мы можем вызвать метод **openOrCreateDatabase()**

Например, создадим базу данных “app.db”:

```
SQLiteDatabase db =
getBaseContext().openOrCreateDatabase("app.db",
MODE_PRIVATE, null);
```

SQLiteOpenHelper

Библиотека Android содержит абстрактный класс **SQLiteOpenHelper**, с помощью которого можно создавать, открывать и обновлять базы данных. При реализации этого вспомогательного класса от вас скрывается логика, на основе которой принимается решение о создании или обновлении базы данных перед ее открытием и другое.

Обязательные методы SQLiteOpenHelper

Класс **SQLiteOpenHelper** содержит два обязательных абстрактных метода:

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

Методы SQLiteOpenHelper

Также используются другие методы класса:

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

Наследник SQLiteOpenHelper

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.

```
class DBHelper extends SQLiteOpenHelper {

  public DBHelper(Context context) {
    // конструктор суперкласса
    super(context, "myDB", null, 1);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    Log.d(LOG_TAG, "--- onCreate database ---");
    // создаем таблицу с полями
    db.execSQL("create table mytable ("
        + "id integer primary key autoincrement,"
        + "name text,"
        + "email text" + ");");
  }

  @Override
  public void onUpgrade(SQLiteDatabase db,
        int oldVersion, int newVersion) {
  }
}
```

Наследник SQLiteOpenHelper

Создаем экземпляр нашего вспомогательного класса DBHelper в основном коде Activity:

```
DBHelper dbHelper;

/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

 // создаем объект для создания и управления версиями БД
 dbHelper = new DBHelper(this);
}
```

Получение базы данных

Вызовите метод **getReadableDatabase()** или **getWritableDatabase()**, чтобы открыть и вернуть экземпляр базы данных:

```
SQLiteDatabase db;
try {
    db = dbHelper.getWritableDatabase();
}
catch (SQLiteException ex){
    db = dbHelper.getReadableDatabase();
}
```

SQLiteDatabase

Для управления базой данных SQLite существует класс **SQLiteDatabase**. В этом классе определены методы:

insert() – для добавления
update() для изменения
delete() – для удаления
query() – для чтения

Кроме того, метод **execSQL()** позволяет выполнять любой допустимый код на языке SQL применимо к таблицам базы данных, если вы хотите провести эти операции вручную.

Insert

Для вставки новой записи в базу данных SQLite используется метод **insert()**:

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

Метод **insert()** возвращает идентификатор _id вставленной строки или -1 в случае ошибки.

Параметры Insert

В методе **insert()** три параметра:

1) **table** — имя таблицы
2) **nullColumnHack** —значение null для столбца в случае пустых строк;
3) **values** — карта отображений (класс **Map** и его наследники), передаваемая клиентом контент-провайдера, которая содержит пары ключ-значение. Ключи в карте должны быть названиями столбцов таблицы, значения — вставляемыми данными.

ContentValues

Для создания новой строки понадобится объект **ContentValues**, точнее, его метод **put()**, чтобы обеспечить данными каждый столбец. Объект **ContentValues** представляет собой пару **name/value** данных. Вставьте новую строку, передавая в метод **insert()**, вызванный в контексте нужной нам базы данных, имя таблицы и объект **ContentValues.**

ContentValues

Для создания новой строки понадобится объект **ContentValues**, точнее, его метод **put()**, чтобы обеспечить данными каждый столбец. Объект **ContentValues** представляет собой пару **name/value** данных. Вставьте новую строку, передавая в метод **insert()**, вызванный в контексте нужной нам базы данных, имя таблицы и объект **ContentValues.**

Insert. Шаблон действий

```
// Создайте новую строку со значениями для вставки.
ContentValues newValues = new ContentValues();

// Задайте значения для каждой строки.
 newValues.put(COLUMN_NAME, newValue);
[ ... Повторите для каждого столбца ... ]

// Вставьте строку в вашу базу данных.
db.insert(DATABASE_TABLE, null, newValues);
```

Insert. Пример

```
// подключаемся к БД
SQLiteDatabase db = dbHelper.getWritableDatabase();

// создаем объект для данных
ContentValues cv = new ContentValues();

// получаем данные из полей ввода
String name = etName.getText().toString();
String email = etEmail.getText().toString();

// подготовим данные для вставки в виде пар
cv.put("name", name);
cv.put("email", email);
// вставляем запись и получаем ее ID
long rowID = db.insert("mytable", null, cv);
```

Update

Для обновления записей в базе данных используют метод **update()**

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

В этом методе два последних параметра формируют SQL-выражение **WHERE**. Метод возвращает число модифицированных строк.

ContentValues

Обновление строк также происходит с помощью класса **ContentValues**.

Создайте новый объект **ContentValues**, используя метод **put()** для вставки значений в каждый столбец, который вы хотите обновить.

Вызовите метод **update()** в контексте базы данных, передайте ему имя таблицы, обновленный объект **ContentValues** и оператор **WHERE**, указывающий на строку (строки), которую нужно обновить.

Update. Шаблон действий

```
// Определите содержимое обновленной строки.
ContentValues newValues= new ContentValues();

// Назначьте значения для каждой строки.
 newValues.put(COLUMN_NAME, newValue);
[ ... Повторите для каждого столбца ... ]

String where = KEY_ID + "=" + rowId;
// Обновите строку с указанным индексом, используя новые значения.
myDatabase.update(DATABASE_TABLE,
newValues, where, null);
```

Update. Пример

```
// подключаемся к БД
SQLiteDatabase db = dbHelper.getWritableDatabase();

// создаем объект для данных
ContentValues cv = new ContentValues();

// получаем данные из полей ввода
String name = etName.getText().toString();
String email = etEmail.getText().toString();
String id = etID.getText().toString();
// подготовим значения для обновления
cv.put("name", name);
cv.put("email", email);
// обновляем по id
int updCount = db.update("mytable", cv, "id = ?",
    new String[] { id });
```

Update. Еще пример

```
myDatabase.update(DATABASE_TABLE,
    murzikValues,
    "NAME = ? OR DESCRIPTION = ?",
    new String[] {"Murzik", "Nice"});
```

Данный код соответствует выражению:

Where NAME = "Murzik" or DESCRIPTION = "Nice".

Delete

Чтобы удалить строку, просто вызовите метод **delete()** в контексте базы данных, указав имя таблицы и оператор **WHERE**. В результате вы получите строки, которые хотите удалить:

int delete (String table, String whereClause, String[] whereArgs)

Delete. Пример

```
// подключаемся к БД
SQLiteDatabase db = dbHelper.getWritableDatabase();
```

```
String id = etID.getText().toString();
// удаляем по id
int delCount = db.delete("mytable", "id = " + id, null);
```

Query

Для чтения данных используют вызов метода **query()**. В метод **query()** передают семь параметров. Если какой-то параметр для запроса вас не интересует, то оставляете *null*:

```
Cursor query (String table,
                String[] columnNames,
                String whereClause,
                String[] selectionArgs,
                String groupBy,
                String having,
                String orderBy )
```

Query. Параметры

table — имя таблицы;

columnNames — список имен возвращаемых полей (массив). При передаче *null* возвращаются все столбцы;

whereClause — параметр, формирующий выражение WHERE (исключая сам оператор WHERE). Значение null возвращает все строки. Например: *_id = 19 and summary = ?*

String[] selectionArgs — значения аргументов фильтра. Вы можете включить ? в "whereClause"". Подставляется в запрос из заданного массива;

Query. Параметры

groupBy - фильтр для группировки, формирующий выражение GROUP BY (исключая сам оператор GROUP BY). Если GROUP BY не нужен, передается null;

having — фильтр для группировки, формирующий выражение HAVING (исключая сам оператор HAVING). Если не нужен, передается null;

orderBy — параметр, формирующий выражение ORDER BY (исключая сам оператор ORDER BY). При сортировке по умолчанию передается null.

Query. Пример1

Чтобы получить все записи из нужных столбцов без условий, достаточно указать имя таблицы в первом параметре и строчный массив во втором. В остальных параметрах оставляем **null**:

```
Cursor cursor = db.query("CAT",
        new String[] {"NAME", "DESCRIPTION"},
        null, null, null, null, null);
```

Пример2. Условие

Для запроса с условием используются третий и четвёртый параметр. Например, нам нужны записи, у которого имя кота будет Барсик:

```
Cursor cursor = db.query("CAT",
        new String[] {"NAME", "DESCRIPTION"},
        "NAME = ?",
        new String[] {"Barsik"},
        null, null, null);
```

Пример3. Сложное условие

```
Cursor cursor = db.query("CAT",
        new String[] {"NAME", "DESCRIPTION"},
        "NAME = ? OR DESCRIPTION = ?",
        new String[] {"Murzik", "Nice"},
        null, null, null);
```

Пример4. Числа

```
Cursor cursor = db.query("CAT",
        new String[] {"NAME", "DESCRIPTION"},
        "_id = ?",
        new String[] {Integer.toString(1)},
        null, null, null);
```

Пример5. Сортировка

Для сортировки используется последний параметр. Нужно указать нужный столбец, по которому будет проводиться сортировка. По алфавиту с буквы А - ASC, наоборот – DESC

```
Cursor cursor = db.query("CAT",
        new String[] {"_id", "NAME", "AGE"},
        null, null, null, null,
        "NAME ASC");
```

Пример6. AVG(), COUNT() и др

Можно использовать функции SQL для составления запросов: AVG(), COUNT(), SUM(), MAX(), MIN() и др.

```
// Возвращает число котов в столбце count
// Аналог SELECT COUNT(_id) AS COUNTER FROM CAT

Cursor cursor = db.query("CAT",
        new String[] {"COUNT(_id) AS counter"},
        null, null, null, null, null);
```

Метод rawQuery()

Также существует метод **rawQuery()**, принимающий сырой SQL-запрос.
Метод **rawQuery()** - это сырой запрос, как есть, т.е. пишется строка запроса, как это обычно делается в SQL.

// Пример:

```
Cursor cursor = db.rawQuery("select * from mytable where _id = ?", new String[] { id });
```

Cursor

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**
- **getFloat**

Методы работы с Cursor

- **moveToFirst** – делает первую запись в Cursor активной и заодно проверяет, есть ли вообще записи в нем.

- **getColumnIndex** - получение порядковых номеров столбцов в Cursor по их именам. Эти номера потом используем для чтения данных в методах getInt и getString.

- **moveToNext** - перебираем все строки в Cursor пока не добираемся до последней.

- **close** - закрываем курсор (освобождаем занимаемые им ресурсы)

Cursor. Пример

```
// делаем запрос всех данных из таблицы mytable, получаем Cursor
Cursor c = db.query("mytable", null, null, null, null, null, null);

// ставим позицию курсора на первую строку выборки
// если в выборке нет строк, вернется false
if (c.moveToFirst()) {

  // определяем номера столбцов по имени в выборке
  int idColIndex = c.getColumnIndex("id");
  int nameColIndex = c.getColumnIndex("name");
  int emailColIndex = c.getColumnIndex("email");

  do {
    // получаем значения по номерам столбцов и пишем все в лог
    Log.d(LOG_TAG,
        "ID = " + c.getInt(idColIndex) +
        ", name = " + c.getString(nameColIndex) +
        ", email = " + c.getString(emailColIndex));
    // переход на следующую строку
    // а если следующей нет (текущая - последняя), то false - выходим из цикла
  } while (c.moveToNext());
} else
  Log.d(LOG_TAG, "0 rows");
c.close();
```

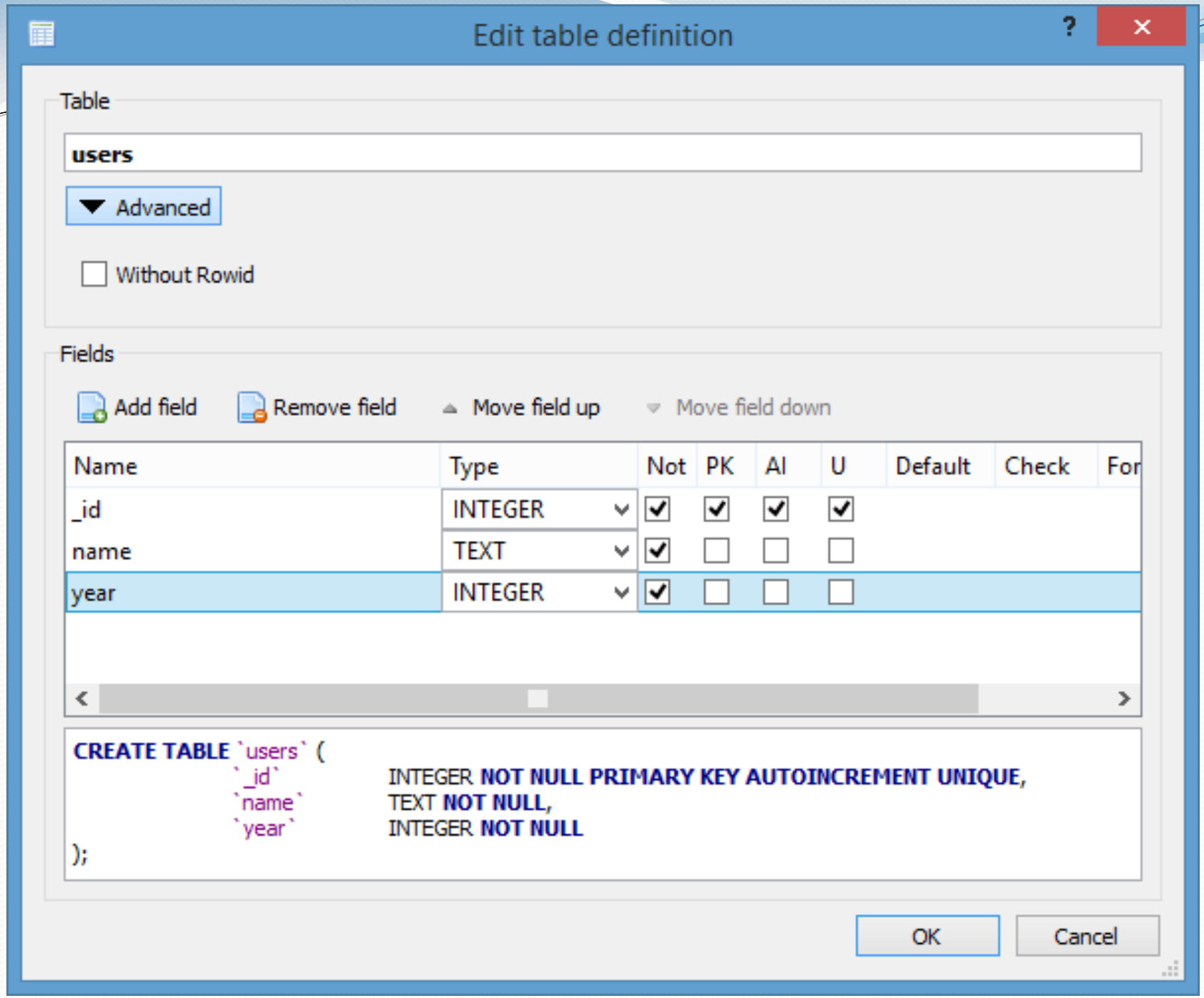
Использование существующей БД SQLite

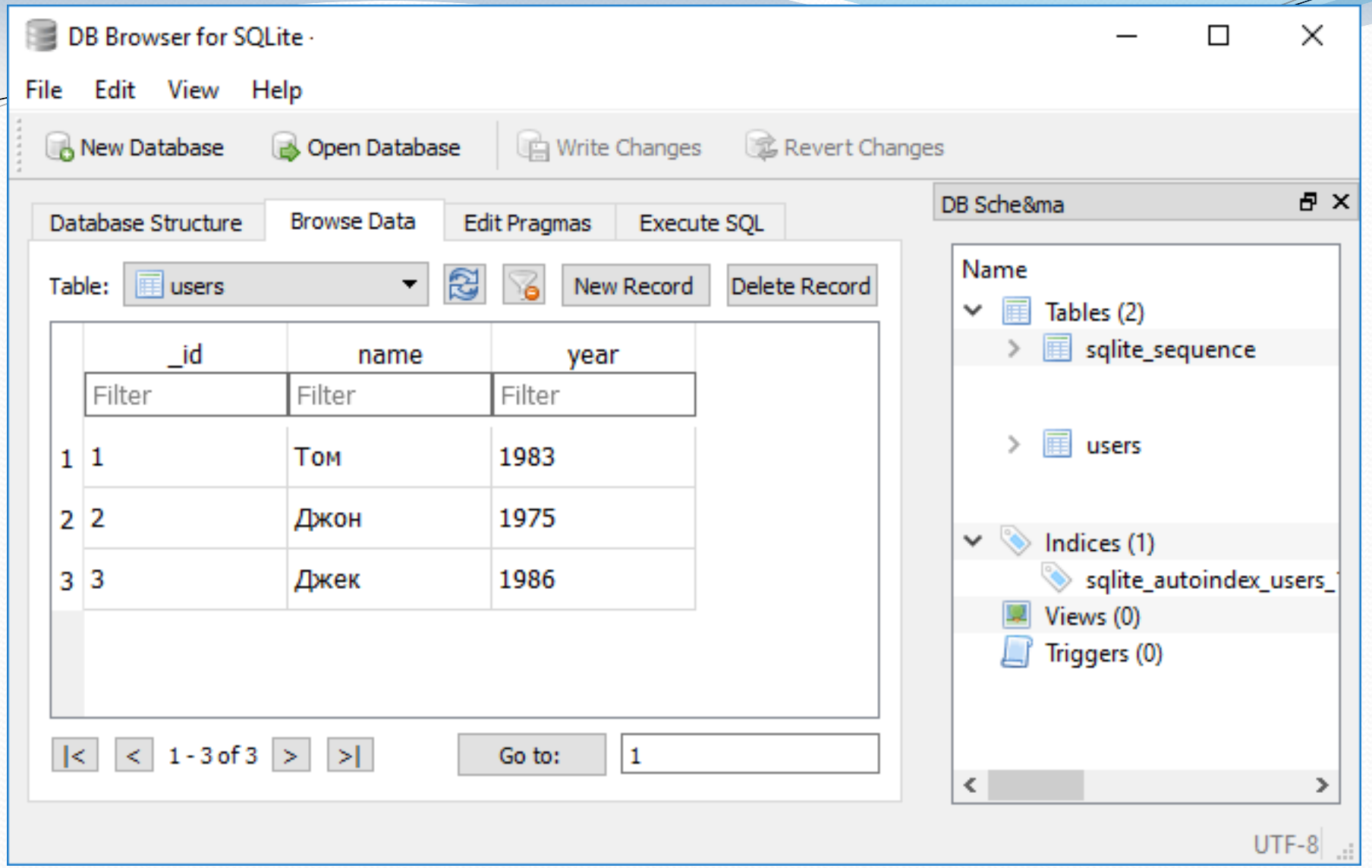
Кроме создания новой базы данных мы также можем использовать уже существующую. Это может быть более предпочтительно, так как в этом случае база данных приложения уже будет содержать всю необходимую информацию.

Sqlitebrowser

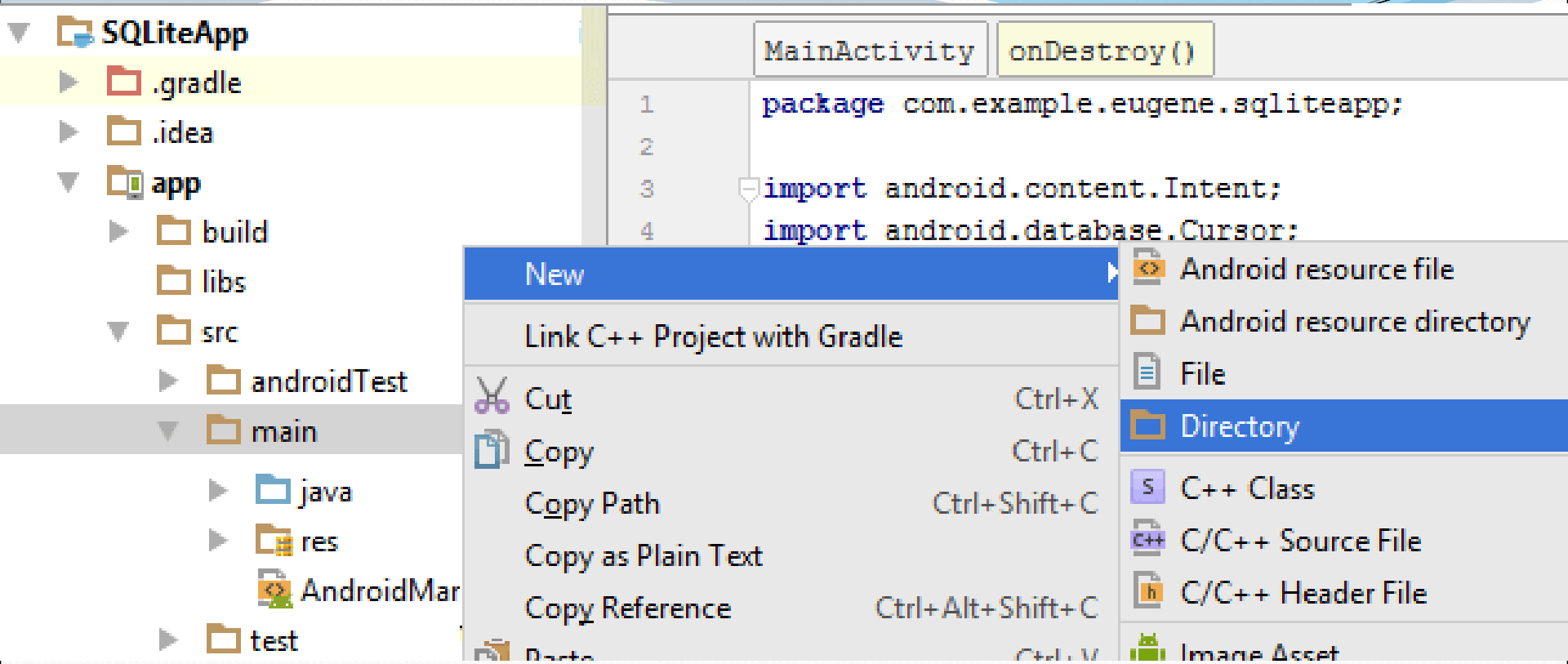
Для начала создадим базу данных SQLite. В этом нам может помочь такой инструмент как Sqlitebrowser. Он бесплатный и доступен для различных операционных систем по адресу http://sqlitebrowser.org/. Хотя можно использовать и другие способы для создания начальной БД.

Sqlitebrowser представляет графический интерфейс для создания базы данных и определения в ней всех необходимых таблиц:

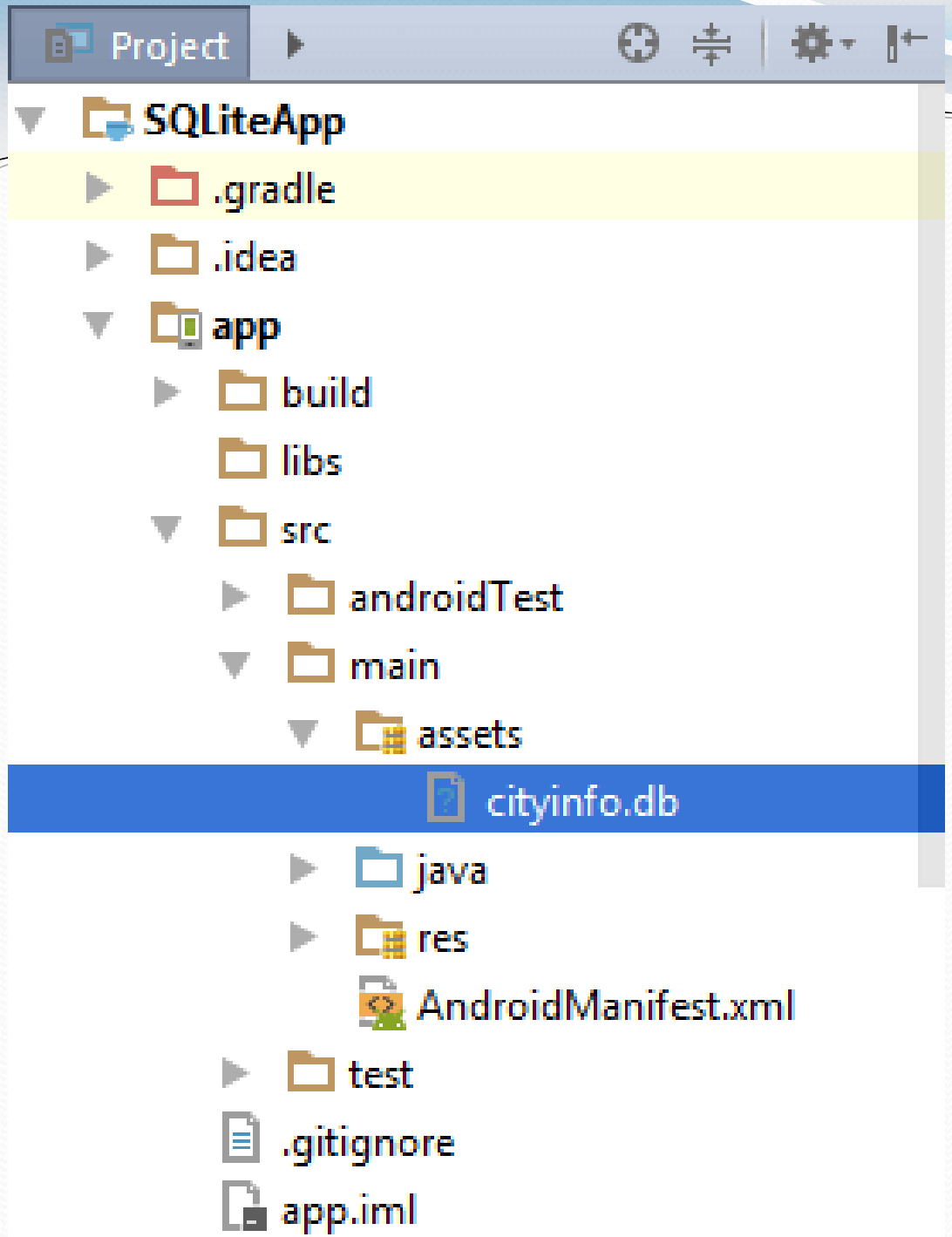

Edit table definition
Table
users
Advanced
Without Rowid
Fields
Add field
Remove field
Move field up
Move field down
Name
Type
Not
PK
AI
U
Default
Check
For
_id
INTEGER
name
TEXT
year
INTEGER
CREATE TABLE `users` (
`_id` INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,
`name` TEXT NOT NULL,
`year` INTEGER NOT NULL
);
OK
Cancel



Добавляем несколько элементов в созданную таблицу



После создания таблицы добавим в проект в Android Studio папку assets, а в папку assets - только что созданную базу данных. Для этого перейдем к полному опеределению проекта, нажмем на папку main правой кнопкой мыши и в меню выберем New -> Directory:



Назовем добавляемую папку **assets** и затем скопируем в нее нашу базу данных

Новый DatabaseHelper

```
class DatabaseHelper extends SQLiteOpenHelper {
    private static String DB_PATH; // полный путь к базе данных
    private static String DB_NAME = "cityinfo.db";
    private static final int SCHEMA = 1; // версия базы данных
    static final String TABLE = "users"; // название таблицы в бд
    // названия столбцов
    static final String COLUMN_ID = "_id";
    static final String COLUMN_NAME = "name";
    static final String COLUMN_YEAR = "year";
    private Context myContext;

    DatabaseHelper(Context context) {
        super(context, DB_NAME, null, SCHEMA);
        this.myContext=context;
        DB_PATH =context.getFilesDir().getPath() + DB_NAME;
    }
    public SQLiteDatabase open()throws SQLException {

        return SQLiteDatabase.openDatabase(DB_PATH, null, SQLiteDatabase.OPEN_
    }
```

Новый DatabaseHelper

Изменим функции onCreate и onUpgrage

```
@Override
public void onCreate(SQLiteDatabase db) {
}
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,  int newVersion)
}
```

Новый DatabaseHelper

Добавляем метод создания базы данных

```
void create_db(){
    InputStream myInput = null;
    OutputStream myOutput = null;
    try {
        File file = new File(DB_PATH);
        if (!file.exists()) {
            this.getReadableDatabase();
            //получаем локальную бд как поток
            myInput = myContext.getAssets().open(DB_NAME);
            // Путь к новой бд
            String outFileName = DB_PATH;

            // Открываем пустую бд
            myOutput = new FileOutputStream(outFileName);

            // побайтово копируем данные
```

Новый DatabaseHelper

Продолжение метода создания базы данных

```
        // побайтово копируем данные
        byte[] buffer = new byte[1024];
        int length;
        while ((length = myInput.read(buffer)) > 0) {
            myOutput.write(buffer, 0, length);
        }

        myOutput.flush();
        myOutput.close();
        myInput.close();
    }
}
catch(IOException ex){
    Log.d("DatabaseHelper", ex.getMessage());
}
```

Путь к внешней БД

По умолчанию база данных будет размещаться во внешнем хранилище, выделяемом для приложения в папке /data/data/[название_пакета]/databases/, и чтобы получить полный путь к базе данных в конструкторе используется выражение:

```
DB_PATH = context.getFilesDir().getPath() +
DB_NAME;
```

Метод onCreate

Метод onCreate() нам не нужен, так как нам не требуется создание встроенной базы данных. Зато здесь определен дополнительный метод create_db(), цель которого копирование базы данных из папки assets в то место, которое указано в переменной DB_PATH.

Кроме этого здесь также определен метод открытия базы данных open() с помощью метода SQLiteDatabase.openDatabase()

Использование БД в Activity

```
databaseHelper = new DatabaseHelper(getApplicationContext());
// создаем базу данных
databaseHelper.create_db();
// открываем подключение
db = databaseHelper.open();
//получаем данные из бд в виде курсора
userCursor =  db.rawQuery("select * from "+ DatabaseHelper.TABLE,
```