



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение  
высшего образования

«МИРЭА – Российский технологический университет»

**РТУ МИРЭА**

---

Институт Информационных технологий

Кафедра Математического обеспечения и стандартизации информационных  
технологий

## **Отчет по практической работе №10**

по дисциплине «Разработка мобильных приложений»

**Выполнил:**

Студент группы ИКБО-20-23

Комисарик М.А.

**Проверил:**

Старший преподаватель кафедры  
МОСИТ

Шешуков Л.С.

Москва 2025 г.

# СОДЕРЖАНИЕ

1 ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ .....	3
1.1 Класс SharedPreferences.....	3
1.2 Основные понятия баз данных .....	5
1.3 Основные команды языка SQL .....	7
1.4 Работа с СУБД в Android.....	13
2 ПРАКТИЧЕСКОЕ ЗАДАНИЕ .....	23
2.1 SharedPreferences .....	23
2.1.1 Разметка.....	23
2.1.2 Реализация.....	24
2.1.3 Тестирование .....	24
2.2 Работа с базами данных.....	26
2.2.1 Разметка.....	26
2.2.2 Реализация.....	28
2.2.3 Тестирование .....	33
ЗАКЛЮЧЕНИЕ .....	36

# 1 ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ

## 1.1 Класс SharedPreferences

В предыдущей практике рассматривался функционал сохранения данных через файлы, что является удобным способом передавать данные между разными активностями и приложениями. Однако, данный способ не является оптимальным в случае с примитивными типами данных, значения которых записывать в отдельный файл не целесообразно, ввиду маленького объема хранимых данных. Для этих целей лучше использовать хранилище SharedPreferences.

Класс SharedPreferences в Android разработке используется для сохранения и получения данных лёгковесных настроек приложения в форме пар ключ-значение. Это позволяет вам сохранять примитивные данные: строки, булевы значения, целые числа и т.д. Эти данные сохраняются в файл на устройстве между сессиями работы приложения, что особенно удобно для сохранения пользовательских настроек, авторизационных данных или любой другой информации, которая должна быть постоянно доступна вне зависимости от того, активно приложение или нет.

Применения класса SharedPreferences:

- хранение пользовательских настроек: идеален для хранения предпочтений пользователя, например, выбранной темы оформления или языка интерфейса;
- хранение состояния приложения: можно использовать для сохранения таких данных, как последняя открытая вкладка или введенные данные в форму, что позволяет пользователям вернуться к тому же состоянию интерфейса после перезапуска приложения;

- хранение небольших данных: эффективное решение для хранения ограниченных объёмов данных без необходимости создавать и управлять базой данных.

Для использования `SharedPreferences`, необходимо получить экземпляр `SharedPreferences` через вызов одного из следующих методов контекста: `getSharedPreferences()` для загрузки настроек по имени файла или `getPreferences()` для работы с настройками конкретной активности (Рисунок 1).

```
// Сохранение данных
SharedPreferences sharedPreferences = getSharedPreferences( name: "myPreferences", MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPreferences.edit();
// Сохранение строкового значения
editor.putString("username", "User123");
editor.putInt("sessionCount", 5);
editor.putBoolean("loggedIn", true);
// Сохранение изменений
editor.apply();
```

**Рисунок 1 – Работа с экземпляром класса `SharedPreferences`**

В методе `getSharedPreferences()` первый параметр метода указывает на название настроек. В данном случае название – "myPreferences". Если настроек с подобным названием нет, то они создаются при вызове данного метода. Второй параметр указывает на режим доступа.

Метод `edit()` возвращает объект `SharedPreferences.Editor`, который используется для редактирования настроек.

При чтении строковых значений, метод `getString (String key, String defValue)` возвращает из настроек значение типа `String`, которое имеет ключ `key`. Если элемента с таким ключом не окажется, то возвращается значение `defValue`, передаваемое вторым параметром (Рисунок 2).

```
// Получение данных
SharedPreferences sharedPreferences = getSharedPreferences( name: "myPreferences", MODE_PRIVATE);
// Чтение строкового значения
String username = sharedPreferences.getString( key: "username", defValue: "defaultUsername");
int sessionCount = sharedPreferences.getInt( key: "sessionCount", defValue: 0);
boolean isLoggedIn = sharedPreferences.getBoolean( key: "loggedIn", defValue: false);
```

**Рисунок 2 – Получение данных из экземпляра класса `SharedPreferences`**

Также можно удалять данные из настроек при помощи метода clear (Рисунок 3).

```
SharedPreferences sharedPreferences = getSharedPreferences("myPreferences", MODE_PRIVATE);  
// Удаление данных  
SharedPreferences.Editor editor = sharedPreferences.edit();  
// Удаление данных по ключу  
editor.remove("key: "username");  
// Удаление всех данных  
editor.clear();  
// Применение изменений  
editor.apply();
```

Рисунок 3 – Удаление данных из настроек

Если попробовать получить данные после того, как они были удалены, то будут получены данные, прописанные по умолчанию.

## 1.2 Основные понятия баз данных

Главный недостаток метода хранения данных, описанного выше — это неструктурированность, что значительно затруднит дальнейшую работу с данными.

Одним из самых популярных способов хранения данных является их структуризация в виде таблиц. Такая концепция основана на структурировании информации в виде таблиц, состоящих из строк и столбцов.

Как это выглядит? Да примерно, как excel-табличка! Есть колонки с заголовками, и информация внутри (Рисунок 4).

last_id	name	email	sort	_lng	action	enable	password
4	Артём Иванович	artem@gmail.com	NULL	NULL	NULL	1	ba3253876aed6bc22d4a6ff53d84
5	Ольга	aa+1@mail.ru	NULL	NULL	NULL	1	4dff4ea340f0a823f15d3f4f01ab62
6	Мария Анатольевна	maria@gmail.com	NULL	NULL	NULL	1	ba3253876aed6bc22d4a6ff53d84
7	Зайка	zaika@gmail.com	NULL	NULL	NULL	1	ba3253876aed6bc22d4a6ff53d84
8	Любимый клиент	client@gmail.com	NULL	NULL	NULL	1	ba3253876aed6bc22d4a6ff53d84

Рисунок 4 – Визуализация настроек

Это одна из самых популярных форм организации данных, обеспечивающая эффективное управление и быстрый доступ к информации. Вот несколько ключевых аспектов этой концепции:

- таблицы: данные хранятся в таблицах, где каждая таблица обычно представляет собой одну сущность или объект (например, клиенты, заказы, продукты);
- строки и столбцы: каждая строка таблицы представляет собой один экземпляр сущности, а столбцы — различные атрибуты этой сущности. Например, в таблице клиентов каждая строка может представлять отдельного клиента, а столбцы — имя, адрес и телефонный номер;
- первичные ключи (Primary Keys): для уникальной идентификации каждой строки в таблице используются первичные ключи. Это обеспечивает возможность точного указания и быстрого поиска любой строки в таблице. Как правило этот столбец генерируется автоматически во избежание дублирования информации и является уникальным;
- внешние ключи (Foreign Keys): с помощью внешних ключей таблицы связываются друг с другом. Например, таблица заказов может содержать внешний ключ, указывающий на таблицу клиентов, что позволяет связать каждый заказ с конкретным клиентом;
- нормализация: данные часто нормализуют для избегания дублирования информации и уменьшения возможностей возникновения ошибок. Нормализация включает разделение данных на несколько таблиц и их связывание через внешние ключи;
- индексация: чтобы ускорить поиск данных, используются индексы. Индексы могут быть созданы для одного или нескольких столбцов, и они позволяют базе данных быстрее находить строки, соответствующие определенным условиям.

Такая концепция широко применяется в реляционных базах данных, где все запросы к ним построены на специальном языке SQL (Structured Query Language, или язык структурированных запросов).

### 1.3 Основные команды языка SQL

SQL (Structured Query Language) — язык общения с базой данных. Для того, чтобы общаться с базой данных для начала необходимо её создать.

Для создания таблиц используется команда **CREATE TABLE**. Команда **CREATE TABLE** (создать таблицу) имеет определённый синтаксис (Рисунок 5).

```
CREATE TABLE название_таблицы
( название_столбца1 тип_данных ограничения_столбца1,
  название_столбца2 тип_данных атрибуты_столбца2 );
```

Рисунок 5 – Синтаксис создания таблицы в SQL

Имя таблицы выполняет роль ее идентификатора в базе данных, поэтому оно должно быть уникальным. Кроме того, оно не должно начинаться на "sqlite\_", поскольку названия таблиц, которые начинаются на "sqlite\_", зарезервированы для внутреннего пользования.

Затем после названия таблицы в скобках перечисляются названия столбцов, их типы данных и атрибуты. В самом конце можно определить атрибуты для всей таблицы. Атрибуты столбцов, а также атрибуты таблицы указывать необязательно.

Например, создадим табличку с данными студента. У каждого студента есть ФИО, а также возраст и пол (Рисунок 6).

```
CREATE TABLE student
( id INTEGER PRIMARY KEY AUTOINCREMENT,
  fio TEXT,
  age INTEGER,
  gender TEXT );
```

Рисунок 6 – Создание таблицы student

**AUTOINCREMENT** – атрибут, который говорит, что идентификатор будет автоматически увеличиваться на 1.

Если мы повторно выполним выше определенную sql-команду для создания таблицы student, то мы столкнемся с ошибкой – ведь мы уже создали таблицу с таким названием. Но могут быть ситуации, когда мы можем точно не знать или быть не уверены, есть ли в базе данных такая таблица (например, когда мы пишем приложение на каком-нибудь языке программирования и используем базу данных, которая не нами создана). И чтобы избежать ошибки, с помощью выражения **IF NOT EXISTS** мы можем задать создание таблицы, если она не существует (Рисунок 7).

```
CREATE TABLE IF NOT EXISTS student
( id INTEGER PRIMARY KEY AUTOINCREMENT,
  fio TEXT,
  age INTEGER,
  gender TEXT)
```

Рисунок 7 – Реализация устранения ошибки при повторном создании существующей таблицы

Если таблицы нет, она будет создана. Если она есть, то никаких действий не будет производиться, и ошибки не возникнет.

Таблица для хранения данных о студентах создана, теперь необходимо заполнить её данными.

Для добавления данных в SQLite применяется команда **INSERT**, которая имеет определённый синтаксис (Рисунок 8).

```
INSERT INTO имя_таблицы [(столбец1, столбец2, ... столбец N)] VALUES
(значение1, значение2, ... значениеN)
```

Рисунок 8 – Синтаксис выражения INSERT INTO

После выражения **INSERT INTO** (вставить в) в скобках можно указать список столбцов через запятую, в которые надо добавлять данные, и в конце после слова **VALUES** (значения) в скобках перечисляют добавляемые для столбцов значения.



Добавим новых студентов в нашу таблицу (Рисунок 9).

```
INSERT INTO student (fio, age, gender) VALUES ('Муравьёва Екатерина Андреевна', 25, 'женский');
```

Рисунок 9 – Добавление элемента в таблицу

После названия таблицы указаны столбцы, в которые мы хотим выполнить добавление данные – (фio, возраст, пол). После оператора **VALUES** указаны значения для этих столбцов. Значения будут передаваться столбцам по позиции. То есть столбцу fio передается строка "Муравьёва Екатерина Андреевна", столбцу age – число 25, а столбцу gender строка "женский". И после успешного выполнения данной команды в таблице появится новая строка.

Стоит отметить, что при добавлении данных необязательно указывать значения абсолютно для всех столбцов таблицы. Например, в примере выше не указано значение для столбца id, поскольку для данного столбца значение будет автоматически генерироваться.

Также можно было бы не указывать названия столбцов (Рисунок 10).

```
INSERT INTO student VALUES (1, 'Муравьёва Екатерина Андреевна', 25, 'женский');
```

Рисунок 10 – Добавление элемента без строгого указания столбцов

Однако в этом случае потребовалось бы указать значения для всех его столбцов, в том числе для столбца id. Причем значения передавались столбцам в том порядке, в котором они идут в таблице.

Теперь в таблице со студентами есть первая запись! Но, этого недостаточно. Добавим еще несколько студентов, чтобы можно было выполнять с данными различные операции (Рисунок 11).

id [PK] integer	fio text	age integer	gender text
1	Муравьёва Екатерина Андреевна	25	женский
2	Иванов Иван Иванович	24	мужской
3	Новичков Дмитрий Евгеньевич	22	мужской
4	Овчинникова Мария Андреевна	24	женский

Рисунок 11 – Расширенная версия таблицы

Для получения данных в SQLite применяется команда **SELECT**. В упрощенном виде она имеет конкретный синтаксис (Рисунок 12).

```
SELECT список_столбцов FROM имя_таблицы
```

Рисунок 12 – Синтаксис команды SELECT

Нередко при получении данных из БД выбираются только те данные, которые соответствуют некоторому определенному условию. Для фильтрации данных в команде **SELECT** применяется оператор **WHERE**, после которого указывается условие (Рисунок 13).

```
SELECT список_столбцов FROM имя_таблицы WHERE условие
```

Рисунок 13 – Указание условия при использовании команды SELECT

Описать контекст рисунка 14 можно следующим образом:

- select — выбери мне такие-то колонки;
- from — из такой-то таблицы базы;
- where — такую-то информацию.

Например, я хочу получить информацию по всем студентам, которым 24 года. Составляю в уме ТЗ: дай мне всю информацию по студентам, у которых возраст = 24. Переделываю в SQL (Рисунок 14).

```
select * from student where age = 24;
```

Рисунок 14 – Конкретизация условия WHERE

Символ \* означает, будут выбраны все колонки (можно выбирать конкретные, а можно сразу все) (Рисунок 15).

1	select * from student where age=24;	Команда
Data output   Сообщения   Notifications		
Результат		
	id [PK] integer	fio text
1	2	Иванов Иван Иванович
2	4	Овчинникова Мария Андреевна

Рисунок 15 – Результат конкретизированного поиска

Если бы была не база данных, а простые excel-файлы, тогда то же действие было бы:

- открыть файл с нужными данными (student);
- поставить фильтр на колонку «Возраст» — 24.

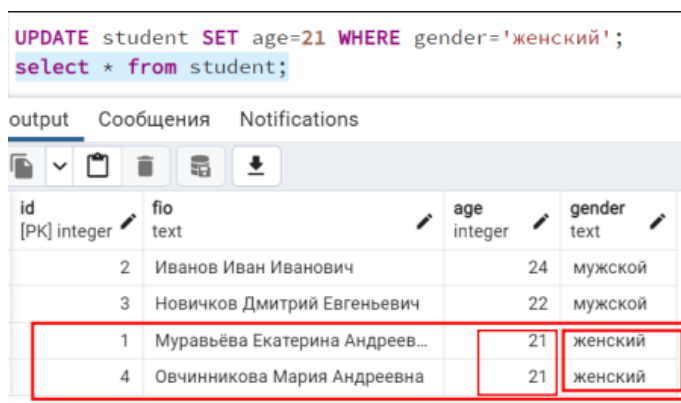
То есть нам в любом случае надо знать название таблицы, где лежат данные, и название колонки, по которой фильтруем. Это не что-то страшное, что есть только в базе данных. То же самое есть в простом экселе.

Для обновления данных в SQLite применяется команда **UPDATE** (обновить). Можно также конкретизировать обновляемые строки с помощью выражения **WHERE**. Тогда команда **UPDATE** имеет конкретный синтаксис (Рисунок 16).

```
UPDATE название_таблицы SET столбец1=значение1, столбец2=значение2,  
WHERE условие_обновления
```

Рисунок 16 – Синтаксис команды UPDATE

Например, возьмем ранее созданную таблицу student. Для всех студентов, которые имеют женский пол, установим возраст 21 (Рисунок 17).



```
UPDATE student SET age=21 WHERE gender='женский';  
select * from student;
```

id	fio	age	gender
[PK] integer	text	integer	text
2	Иванов Иван Иванович	24	мужской
3	Новичков Дмитрий Евгеньевич	22	мужской
1	Муравьёва Екатерина Андреев...	21	женский
4	Овчинникова Мария Андреевна	21	женский

Рисунок 17 – Установка конкретного возраста в таблице

Команда **DELETE** удаляет данные из БД. Она имеет конкретный формальный синтаксис (Рисунок 18).

```
DELETE FROM название_таблицы WHERE условие_удаления
```

Рисунок 18 – Синтаксис команды DELETE

Удалим из нашей таблицы со студентами тех, кому больше 23 лет (Рисунок 19).

The screenshot shows a database interface with a query editor at the top containing the SQL commands: `DELETE FROM student WHERE age>23;` and `select * from student;`. Below the editor are tabs for 'output', 'Сообщения' (Messages), and 'Notifications'. The 'output' tab is active, displaying a table of student data. A red box highlights the table and a red message at the bottom stating 'Запись удалилась из базы данных' (Record deleted from the database).

id	fio	age	gender
3	Новичков Дмитрий Евгеньевич	22	мужской
1	Муравьёва Екатерина Андреев...	21	женский
4	Овчинникова Мария Андреевна	21	женский

Запись удалилась из базы данных

Рисунок 19 – Результат удаления записей

Если необходимо вовсе удалить все строки вне зависимости от условия, то условие можно не указывать (Рисунок 20).

The screenshot shows a SQL statement: `DELETE FROM название_таблицы`, where 'название\_таблицы' is underlined with a red squiggly line, indicating it is a placeholder for a table name.

Рисунок 20 – Синтаксис удаления всех строк из таблицы

По аналогии с созданием таблицы, если мы попытаемся удалить таблицу, которая не существует, то мы столкнемся с ошибкой. В этом случае опять же с помощью операторов **IF EXISTS** проверять наличие таблицы перед удалением (Рисунок 21).

The screenshot shows a database interface with a query editor at the top containing the SQL command: `1 DROP TABLE IF EXISTS student;`. Below the editor are tabs for 'Data output', 'Сообщения' (Messages), and 'Notifications'. The 'Сообщения' tab is active, displaying a message in Russian: 'ЗАМЕЧАНИЕ: таблица "student" не существует, пропускается DROP TABLE' (NOTE: table "student" does not exist, DROP TABLE is skipped).

1	SQL
1	<code>DROP TABLE IF EXISTS student;</code>

ЗАМЕЧАНИЕ: таблица "student" не существует, пропускается DROP TABLE

Рисунок 21 – Способ избегания ошибки при удалении несуществующей таблицы

## 1.4 Работа с СУБД в Android

В Android разработке для управления базой данных часто используется SQLite, встроенная легковесная система управления базами данных (СУБД), которая поддерживает большинство функций SQL. Для взаимодействия с базой данных через SQLite можно использовать Cursor — интерфейс, который предоставляет случайный доступ к результатам запроса к базе данных.

Основную функциональность по работе с базами данных предоставляет пакет `android.database`. Функциональность непосредственно для работы с SQLite находится в пакете `android.database.sqlite`.

База данных в SQLite представлена классом `android.database.sqlite.SQLiteDatabase`. Он позволяет выполнять запросы к бд, выполнять с ней различные манипуляции.

Класс `android.database.sqlite.SQLiteCursor` предоставляет запрос и позволяет возвращать набор строк, которые соответствуют этому запросу.

Класс `android.database.sqlite.SQLiteQueryBuilder` позволяет создавать SQLзапросы.

Сами sql-выражения представлены классом `android.database.sqlite.SQLiteStatement`, которые позволяют с помощью плейсхолдеров вставлять в выражения динамические данные.

Класс `android.database.sqlite.SQLiteOpenHelper` позволяет создать базу данных со всеми таблицами, если их еще не существует.

В качестве примера для создания базы данных будет рассмотрено приложение для записи контактов телефонной книги, где будет одна сущность: Контакт. У каждого контакта есть имя и номер телефона. И конечно же первичный ключ, которым будет выступать идентификатор.

Для начала создадим класс, где будут храниться данные о контактах (Рисунок 22).

```

no usages
public class Contact {
    3 usages
    private int id; //идентификатор
    3 usages
    private String name; //имя
    3 usages
    private String phone; //номер телефона
    no usages
    public Contact(int id, String name, String phone) {
        this.id = id;
        this.name = name;
        this.phone = phone;
    }
    no usages
    public int getId() {
        return id;
    }
    no usages
    public void setId(int id) {
        this.id = id;
    }
    no usages
    public String getName() {
        return name;
    }
    no usages
    public void setName(String name) {
        this.name = name;
    }
    no usages
    public String getPhone() {
        return phone;
    }
    no usages
    public void setPhone(String phone) {
        this.phone = phone;
    }
}

```

**Рисунок 22 – Описание класса Contract**

Далее необходимо создать отдельный класс помощника DatabaseHelper (Рисунок 23) для работы с базой данных, наследуя SQLiteOpenHelper, переопределив как минимум два его метода: onCreate() и onUpgrade(). В этом классе будут определены методы для создания и обновления базы данных.

```

no usages
public class DatabaseHelper extends SQLiteOpenHelper{

    // Информация о базе данных
    1 usage
    private static final String DATABASE_NAME = "ExampleDB";
    1 usage
    private static final int DATABASE_VERSION = 1;

    // Информация о таблице
    2 usages
    public static final String TABLE_NAME = "users";
    1 usage
    public static final String COLUMN_ID = "_id";
    1 usage
    public static final String COLUMN_FIO = "fio";
    1 usage
    public static final String COLUMN_EMAIL = "email";

    // SQL запрос для создания таблицы
    1 usage
    private static final String CREATE_TABLE =
        "CREATE TABLE " + TABLE_NAME + " (" +
            COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
            COLUMN_FIO + " TEXT, " +
            COLUMN_EMAIL + " TEXT);";
no usages
    public DatabaseHelper(Context context) {
        super(context, DATABASE_NAME, factory: null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_TABLE);
    }

    10 usages
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
        onCreate(db);
    }
}

```

Рисунок 23 – Описание класса DatabaseHelper

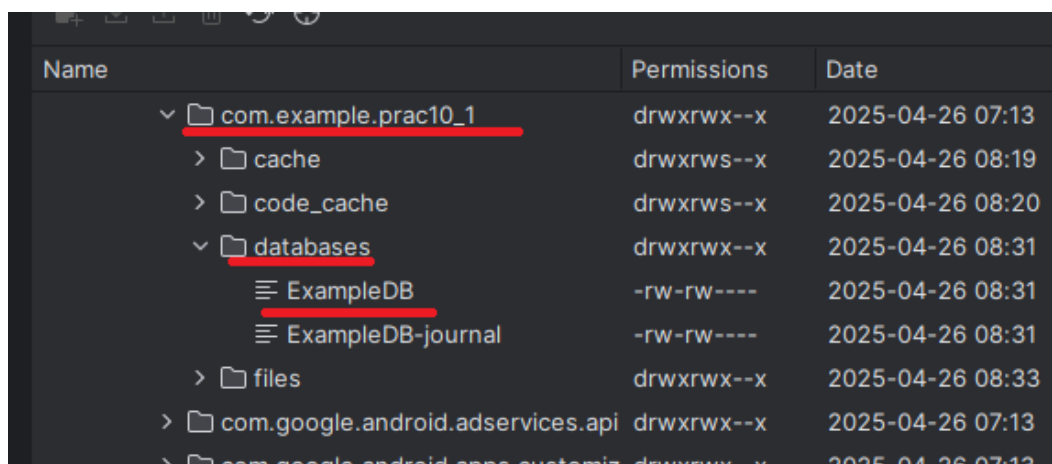
Метод onCreate() вызывается при попытке доступа к базе данных, но когда еще эта база данных не создана.

Для выполнения запроса к базе данных можно использовать метод `execSQL`, в который передается SQL-выражение.

Метод `onUpgrade()` вызывается, когда необходимо обновление схемы базы данных. Здесь можно пересоздать ранее созданную базу данных в `onCreate()`, установив соответствующие правила преобразования от старой бд к новой.

В данном случае для примера использован примитивный подход с удалением предыдущей базы данных с помощью sql-выражения `DROP` и последующим ее созданием. Но в реальности если вам будет необходимо сохранить данные, этот метод может включать более сложную логику – добавления новых столбцов, удаление ненужных, добавление дополнительных данных и так далее.

База данных с таблицей созданы. Теперь необходимо её наполнить. Созданную базу данных можно посмотреть через `Device File Explorer` в файлах `/data/data/название_пакета/databases` (Рисунок 24).



Name	Permissions	Date
com.example.prac10_1	drwxrwx--x	2025-04-26 07:13
> cache	drwxrws--x	2025-04-26 08:19
> code_cache	drwxrws--x	2025-04-26 08:20
> databases	drwxrwx--x	2025-04-26 08:31
ExampleDB	-rw-rw----	2025-04-26 08:31
ExampleDB-journal	-rw-rw----	2025-04-26 08:31
> files	drwxrwx--x	2025-04-26 08:33
> com.google.android.adservices.api	drwxrwx--x	2025-04-26 07:13
> com.google.android.apps.custom...	drwxrwx--x	2025-04-26 07:13

**Рисунок 24 – Расположение созданной базы данных**

Для выполнения операций по вставке, обновлению и удалению данных `SQLiteDatabase` имеет методы `insert()`, `update()` и `delete()`.

Чтобы получить объект базы данных, надо использовать метод `getReadableDatabase()` (получение базы данных для чтения) или `getWritableDatabase()` (запись данных в БД). Так как в данном случае мы будем записывать данные в бд, то воспользуемся вторым методом (Рисунок 25).



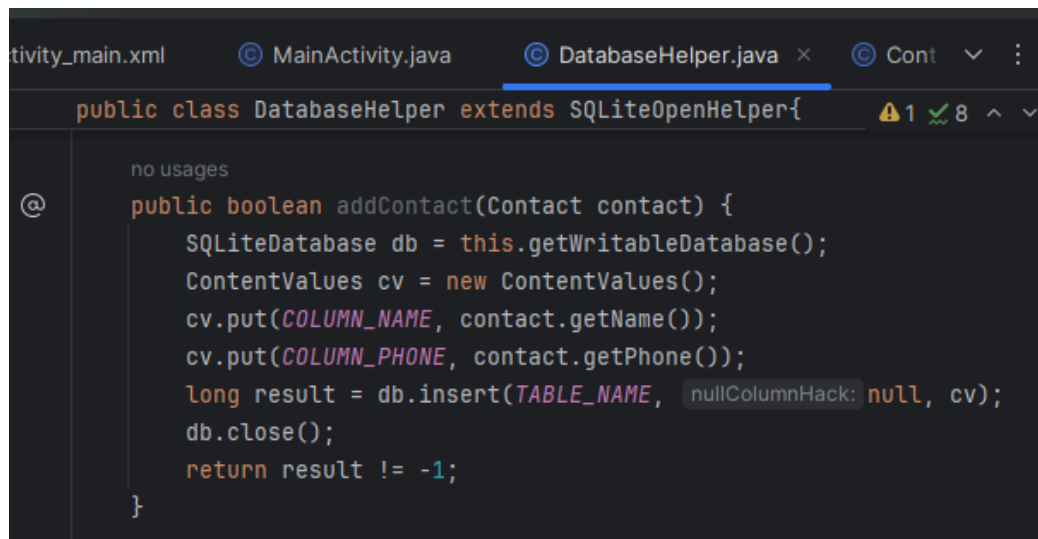


Рисунок 25 – Описание метода `addContact()`

Для добавления или обновления нам надо создать объект `ContentValues`. Данный объект представляет словарь, который содержит набор пар "ключ-значение". Для добавления в этот словарь нового объекта применяется метод `put`. Первый параметр метода – это ключ, а второй – значение.

Метод `insert()` принимает название таблицы, объект `ContentValues` с добавляемыми значениями. Второй параметр является необязательным: он передает столбец, в который надо добавить значение `NULL`.

После завершения работы с базой данных мы закрываем все связанные объекты с помощью метода `close()` (Рисунок 26).

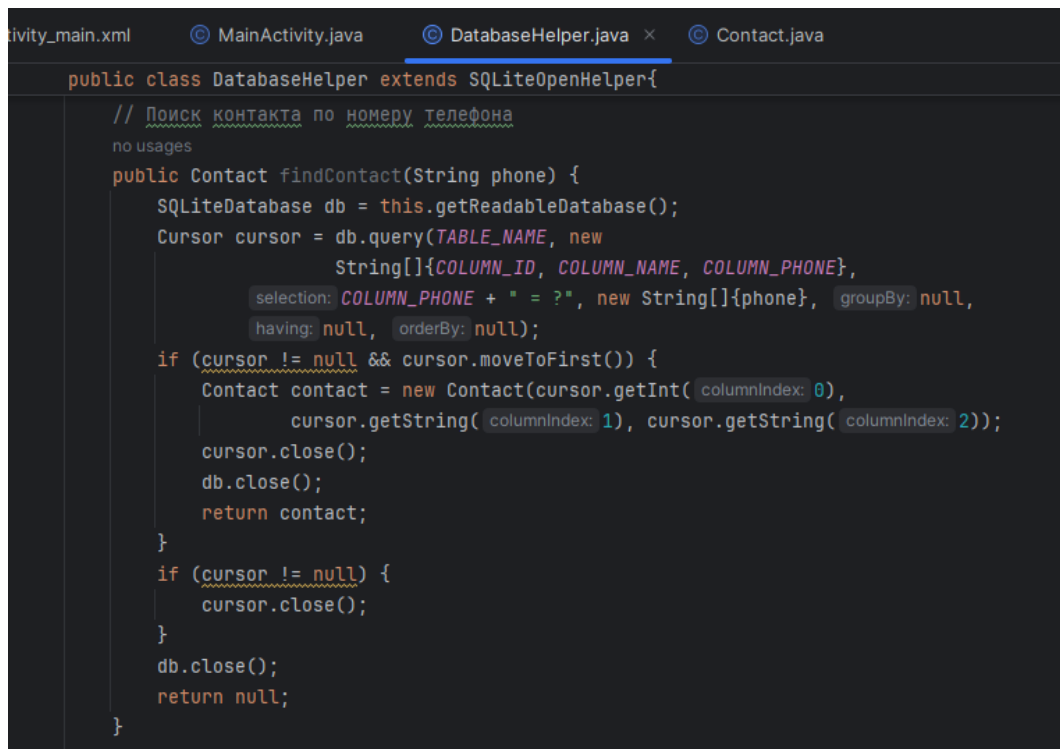


Рисунок 26 – Описание метода `deleteContact()`

В метод `delete()` передается название таблицы, а также столбец, по которому происходит удаление, и его значение. В качестве критерия можно

выбрать несколько столбцов, поэтому третьим параметром идет массив. Знак вопроса "?" обозначает параметр, вместо которого подставляется значение из третьего параметра.

При поиске нужного номера телефона нам необходимо не записывать данные в БД, а читать из нее, с помощью метода `getReadableDatabase()` (Рисунок 27).



```
public class DatabaseHelper extends SQLiteOpenHelper{

    // Поиск контакта по номеру телефона
    no usages
    public Contact findContact(String phone) {
        SQLiteDatabase db = this.getReadableDatabase();
        Cursor cursor = db.query(TABLE_NAME, new
            String[]{COLUMN_ID, COLUMN_NAME, COLUMN_PHONE},
            selection: COLUMN_PHONE + " = ?", new String[]{phone},
            having: null, orderBy: null);
        if (cursor != null && cursor.moveToFirst()) {
            Contact contact = new Contact(cursor.getInt( columnIndex: 0),
                cursor.getString( columnIndex: 1), cursor.getString( columnIndex: 2));
            cursor.close();
            db.close();
            return contact;
        }
        if (cursor != null) {
            cursor.close();
        }
        db.close();
        return null;
    }
}
```

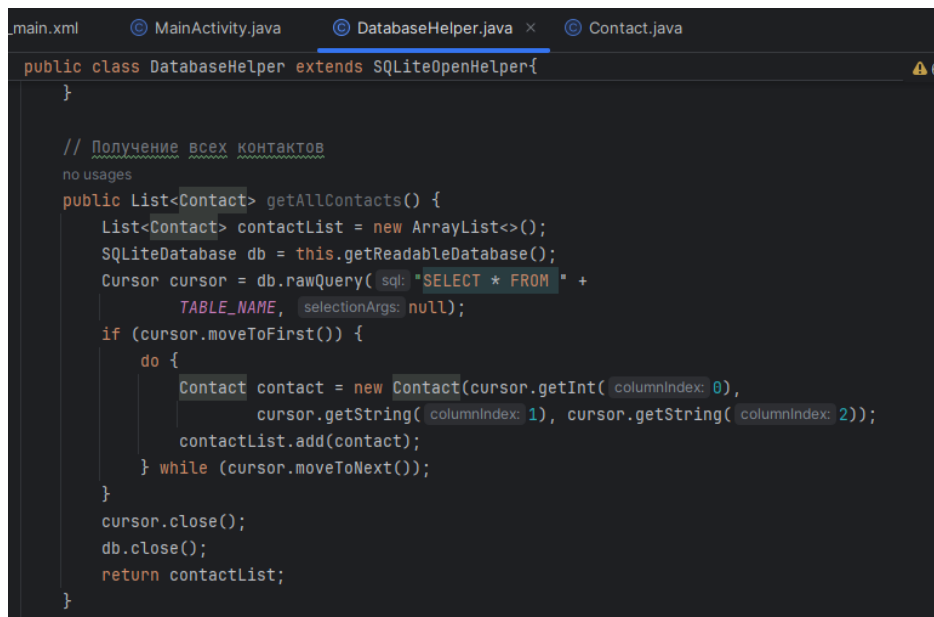
**Рисунок 27 – Описание метода поиска контакта в базе данных**

Класс `Cursor` предлагает ряд методов для управления выборкой, в частности:

- методы `moveToFirst()` и `moveToNext()` позволяют переходить к первому и к следующему элементам выборки;
- методы `get*(columnIndex)` (например, `getLong()`, `getString()`) позволяют по индексу столбца обратиться к данному столбцу текущей строки.

После завершения работы курсор должен быть закрыт методом `close()`.

Можно получить сразу все данные из базы данных, отобразив их в список (Рисунок 28).



```

main.xml  MainActivity.java  DatabaseHelper.java  Contact.java
public class DatabaseHelper extends SQLiteOpenHelper{
}

// Получение всех контактов
no usages
public List<Contact> getAllContacts() {
    List<Contact> contactList = new ArrayList<>();
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = db.rawQuery("SELECT * FROM " +
        TABLE_NAME, selectionArgs: null);
    if (cursor.moveToFirst()) {
        do {
            Contact contact = new Contact(cursor.getInt( columnIndex: 0),
                cursor.getString( columnIndex: 1), cursor.getString( columnIndex: 2));
            contactList.add(contact);
        } while (cursor.moveToNext());
    }
    cursor.close();
    db.close();
    return contactList;
}

```

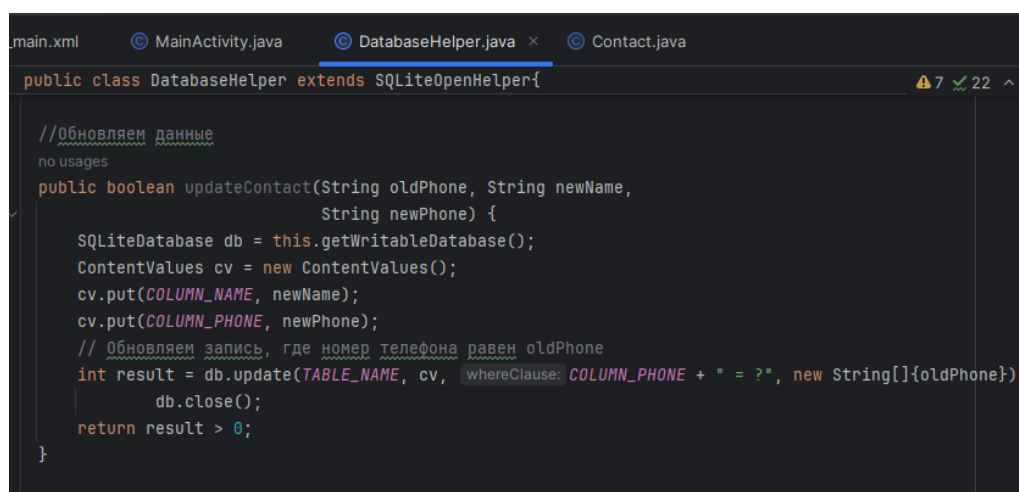
**Рисунок 28 – Описание метода поиска всех возможных контактов в базе данных**

Метод `rawQuery()` возвращает объект `Cursor`, с помощью которого мы можем извлечь полученные данные.

Возможна ситуация, когда в базе данных не будет объектов, и для этого методом `moveToFirst()` пытаемся переместиться к первому объекту, полученному из базы данных. Если этот метод возвратит значение `false`, значит запрос не получил никаких данных из базы данных.

Вызовом `moveToNext()` перемещаемся в цикле `while` последовательно по всем объектам.

Также необходимо описать метод обновления контакта в базе данных (Рисунок 29).



```

main.xml  MainActivity.java  DatabaseHelper.java  Contact.java
public class DatabaseHelper extends SQLiteOpenHelper{
}

//Обновляем данные
no usages
public boolean updateContact(String oldPhone, String newName,
    String newPhone) {
    SQLiteDatabase db = this.getWritableDatabase();
    ContentValues cv = new ContentValues();
    cv.put(COLUMN_NAME, newName);
    cv.put(COLUMN_PHONE, newPhone);
    // обновляем запись, где номер телефона равен oldPhone
    int result = db.update(TABLE_NAME, cv, whereClause: COLUMN_PHONE + " = ?", new String[]{oldPhone});
    db.close();
    return result > 0;
}

```

**Рисунок 29 – Описание метода обновления контакта в базе данных**

Затем создаем главную активность, на экране которой будет динамический список со всеми контактами. При добавлении, изменении или удалении данных, список на экране будет изменяться (Рисунки 30-32).

```
© MainActivity.java × © ContactAdapter.java </> item_contact.xml build.gradle.kts (prac10_1) build.gradle

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        EdgeToEdge.enable(this);
        setContentView(R.layout.activity_main);
        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main), (v, insets) -> {
            Insets systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars());
            v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom);
            return insets;
        });

        EditText nameInput = findViewById(R.id.name_input);
        EditText phoneInput = findViewById(R.id.phone_input);
        Button saveButton = findViewById(R.id.save_button);
        Button deleteButton = findViewById(R.id.delete_button);
        Button findButton = findViewById(R.id.find_button);
        RecyclerView contactsList =
            findViewById(R.id.contacts_list);
        DatabaseHelper dbHelper = new DatabaseHelper(context: this);
        List<Contact> contacts = dbHelper.getAllContacts();
        ContactAdapter adapter = new ContactAdapter(contacts);
        contactsList.setLayoutManager(new
            LinearLayoutManager(context: this));
        contactsList.setAdapter(adapter);

        //Прописываем логику для сохранения нового контакта
        saveButton.setOnClickListener(v -> {
            String name = nameInput.getText().toString();
            String phone = phoneInput.getText().toString();
            if (dbHelper.addContact(new Contact(id: 0, name, phone)))
            {
                contacts.add(new Contact(id: 0, name, phone));
                adapter.notifyItemInserted(position: contacts.size() - 1);
                Toast.makeText(context: this, text: "Contact saved successfully!", Toast.LENGTH_SHORT).show();
            } else {
                Toast.makeText(context: this, text: "Failed to save contact",
                    Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

Рисунок 30 – Описание класса MainActivity, часть 1

```

    MainActivity.java    ContactAdapter.java    item_contact.xml    build.gradle.kts (prac10_1)    build.gradle.kts (ap

public class MainActivity extends AppCompatActivity {
    protected void onCreate(Bundle savedInstanceState) {
        //удаляем КОНТАКТ
        deleteButton.setOnClickListener(v -> {
            String phone = phoneInput.getText().toString();
            if (dbHelper.deleteContact(phone)) {
                int position = -1;
                for (int i = 0; i < contacts.size(); i++) {
                    if (contacts.get(i).getPhone().equals(phone))
                    {
                        position = i;
                        contacts.remove(i);
                        break;
                    }
                }
                if (position != -1) {
                    adapter.notifyItemRemoved(position);
                    Toast.makeText(context, this, text: "Contact deleted successfully!", Toast.LENGTH_SHORT).show();
                } else {
                    Toast.makeText(context, this, text: "Contact not found",
                        Toast.LENGTH_SHORT).show();
                }
            } else {
                Toast.makeText(context, this, text: "Failed to delete contact",
                    Toast.LENGTH_SHORT).show();
            }
        });
        //ищем КОНТАКТ по номеру телефона
        findButton.setOnClickListener(v -> {
            String phone = phoneInput.getText().toString();
            Contact foundContact = dbHelper.findContact(phone);
            if (foundContact != null) {
                nameInput.setText(foundContact.getName());
                phoneInput.setText(foundContact.getPhone());
                Toast.makeText(context, this, text: "Contact found: " +
                    foundContact.getName(), Toast.LENGTH_SHORT).show();
            } else {
                Toast.makeText(context, this, text: "Contact not found",
                    Toast.LENGTH_SHORT).show();
            }
        });
    }
}

```

Рисунок 31 – Описание класса MainActivity, часть 2

```

    MainActivity.java    ContactAdapter.java    item_contact.xml    build.gradle.kts (prac10_1)    build.gradle.kts (app)    DatabaseHelper.jav

public class MainActivity extends AppCompatActivity {
    protected void onCreate(Bundle savedInstanceState) {
        findButton.setOnClickListener(v -> {
            foundContact.getName(), Toast.LENGTH_SHORT).show();
        } else {
            Toast.makeText(context, this, text: "Contact not found",
                Toast.LENGTH_SHORT).show();
        }
    }
};

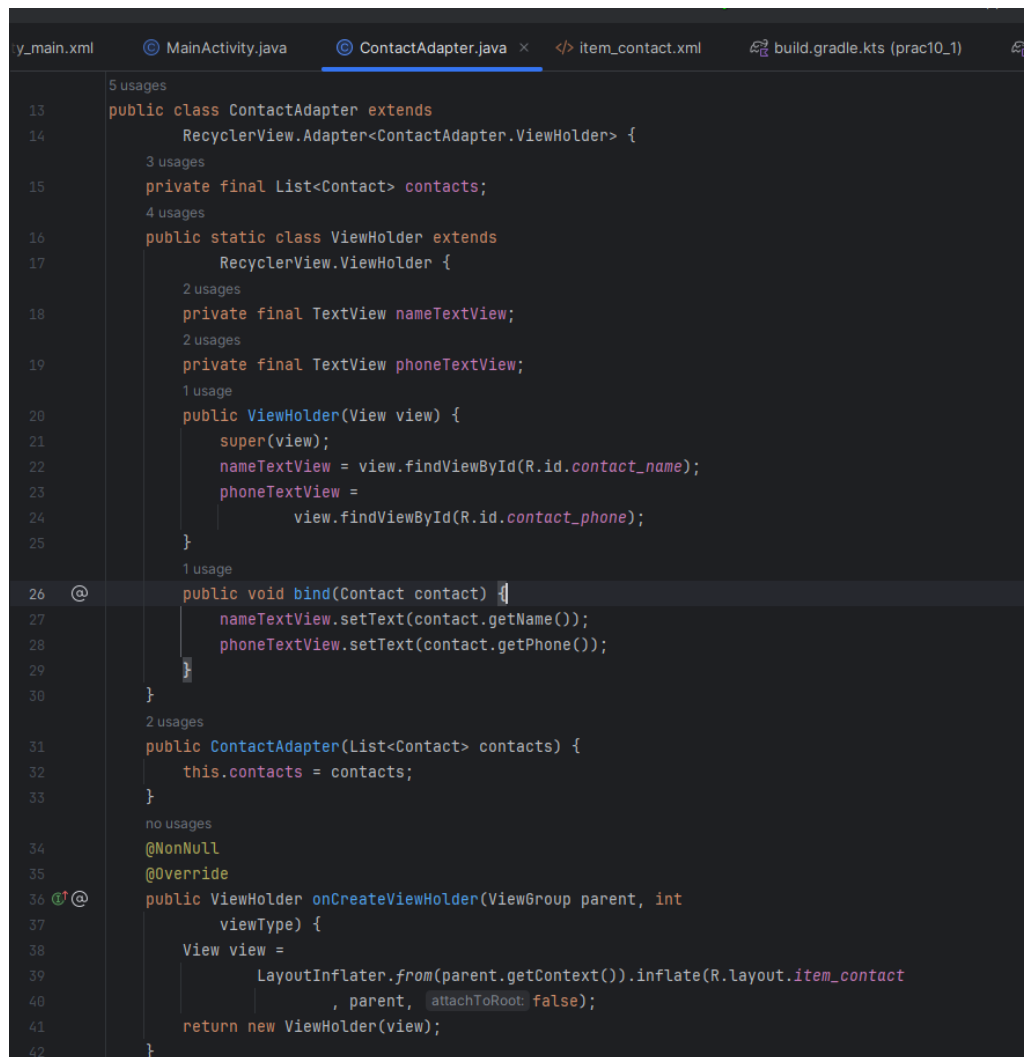
//обновляем данные
Button updateButton = findViewById(R.id.update_button);
updateButton.setOnClickListener(v -> {
    String oldPhone = phoneInput.getText().toString(); //Считаем что это старый номер для поиска
    String newName = nameInput.getText().toString(); //Новое имя для обновления
    String newPhone = phoneInput.getText().toString(); //Новый номер для обновления
    if (dbHelper.updateContact(oldPhone, newName,
        newPhone)) {
        Toast.makeText(context, this, text: "Contact updated successfully!", Toast.LENGTH_SHORT).show();
        // обновляем список и адаптер
        refreshContactsList(dbHelper, contacts, adapter,
            contactsList);
    } else {
        Toast.makeText(context, this, text: "Failed to update contact",
            Toast.LENGTH_SHORT).show();
    }
});

// Метод для обновления списка контактов после изменения в базе данных
1 usage
private void refreshContactsList(DatabaseHelper dbHelper, List<Contact> contacts, ContactAdapter adapter, RecyclerView contactsList) {
    contacts = dbHelper.getAllContacts(); // Загружаем обновленный список
    adapter = new ContactAdapter(contacts);
    contactsList.setAdapter(adapter);
}
}

```

Рисунок 32 – Описание класса MainActivity, часть 3

Так как мы используем RecyclerView, то нам также необходимо прописать код адаптера для обновления данных в списке (Рисунки 33-34).



```
13 public class ContactAdapter extends
14     RecyclerView.Adapter<ContactAdapter.ViewHolder> {
15     3 usages
16     private final List<Contact> contacts;
17     4 usages
18     public static class ViewHolder extends
19         RecyclerView.ViewHolder {
20     2 usages
21     private final TextView nameTextView;
22     2 usages
23     private final TextView phoneTextView;
24     1 usage
25     public ViewHolder(View view) {
26         super(view);
27         nameTextView = view.findViewById(R.id.contact_name);
28         phoneTextView =
29             view.findViewById(R.id.contact_phone);
30     }
31     1 usage
32     public void bind(Contact contact) {
33         nameTextView.setText(contact.getName());
34         phoneTextView.setText(contact.getPhone());
35     }
36     2 usages
37     public ContactAdapter(List<Contact> contacts) {
38         this.contacts = contacts;
39     }
40     no usages
41     @NonNull
42     @Override
43     public ViewHolder onCreateViewHolder(ViewGroup parent, int
44         viewType) {
45         View view =
46             LayoutInflater.from(parent.getContext()).inflate(R.layout.item_contact
47                 , parent, attachToRoot: false);
48         return new ViewHolder(view);
49     }
50 }
```

Рисунок 33 – Описание класса ContactAdapter, часть 1



```
51     @Override
52     public void onBindViewHolder(ViewHolder holder, int position)
53     {
54         holder.bind(contacts.get(position));
55     }
56     @Override
57     public int getItemCount() {
58         return contacts.size();
59     }
60 }
```

Рисунок 34 – Описание класса ContactAdapter, часть 2

Даже если перезапустить приложение, то при открытии будет выведен список сохраненных ранее контактов. Таким образом, получилось готовое приложение для хранения телефонных контактов.

## 2 ПРАКТИЧЕСКОЕ ЗАДАНИЕ

### 2.1 SharedPreferences

#### 2.1.1 Разметка

Создадим проект и добавим в файл разметки activity\_main.xml несколько элементов:

- EditText для ввода имени пользователя;
- Button «Установить имя пользователя» для установки нового значения имени пользователя в SharedPreferences данного приложения;
- Button «Удалить имя пользователя» для удаления имени пользователя из SharedPreferences данного приложения;
- Button «Получить имя пользователя» для получения имени пользователя из SharedPreferences данного приложения;
- TextView для вывода имени пользователя.

Файл разметки представлен на рисунке 35.

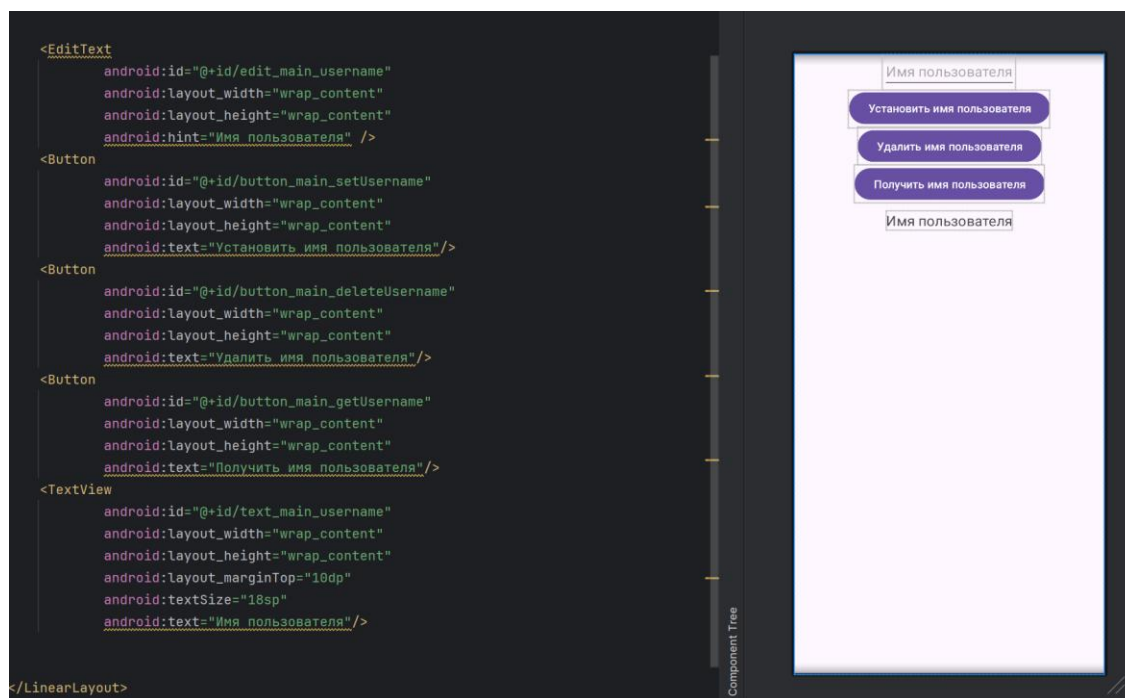


Рисунок 35 – Файл разметки activity\_main.xml

## 2.1.2 Реализация

В методе `onCreate()` класса `MainActivity` происходит привязка нажатия кнопок к соответствующим действиям. Для хранения имени пользователя был использован объект `SharedPreferences`, получаемый с помощью метода `getPreferences()`, то есть принадлежащий данному приложению (Рисунок 36).

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    EdgeToEdge.enable( $this$enableEdgeToEdge: this);
    setContentView(R.layout.activity_main);
    ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main), (v, insets) ->
    {
        Insets systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars());
        v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom);
        return insets;
    });

    SharedPreferences prefs = getPreferences(MODE_PRIVATE);

    EditText editUsername = findViewById(R.id.edit_main_username);

    Button buttonSetUsername = findViewById(R.id.button_main_setUsername);
    buttonSetUsername.setOnClickListener(v -> prefs
        .edit()
        .putString(PREFERENCE_USERNAME, editUsername.getText().toString())
        .apply());
    Button buttonDeleteUsername = findViewById(R.id.button_main_deleteUsername);

    buttonDeleteUsername.setOnClickListener(v -> prefs
        .edit()
        .remove(PREFERENCE_USERNAME)
        .apply());
    Button buttonGetUsername = findViewById(R.id.button_main_getUsername);
    TextView textUsername = findViewById(R.id.text_main_username);
    buttonGetUsername.setOnClickListener(v ->
        textUsername.setText(prefs.getString(PREFERENCE_USERNAME, defValue: "Не задано")));
}
```

Рисунок 36 – Описание метода `onCreate()` класса `MainActivity`

## 2.1.3 Тестирование

Запустим приложение (Рисунок 37).





**Рисунок 37 – Начальный экран приложения по работе с SharedPreferences**

Введем имя «Михаил» и нажмем кнопку «Установить имя пользователя», а затем «Получить имя пользователя», после чего установленное имя отобразится в текстовом поле (Рисунок 38).



**Рисунок 38 – Отображение имени пользователя, полученного с помощью SharedPreferences**

Удалим имя пользователя из SharedPreferences с помощью кнопки «Удалить имя пользователя» и затем нажмем кнопку «Получить имя пользователя», после чего в текстовом поле отобразится, что пользовательское имя не задано (Рисунок 39).



**Рисунок 39 – Удаление имени пользователя из SharedPreferences**

Тестирование прошло успешно.

## **2.2 Работа с базами данных**

### **2.2.1 Разметка**

После создания проекта для работы с базами данных, добавим несколько полей для ввода текста (Рисунок 40), кнопки для выполнения действий (Рисунок 41), а также строчки для вывода результатов запросов к базе данных (Рисунок 42).

```

<EditText
    android:id="@+id/edit_main_id"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:gravity="center"
    android:inputType="number"
    android:hint="ID" />
<TableRow
    android:layout_width="match_parent"
    android:layout_height="48dp"
    android:gravity="center">
    <EditText
        android:id="@+id/edit_main_name"
        android:layout_width="wrap_content"
        android:gravity="center"
        android:hint="Название" />
    <EditText
        android:id="@+id/edit_main_weight"
        android:layout_width="wrap_content"
        android:gravity="center"
        android:hint="Вес" />
    <EditText
        android:id="@+id/edit_main_price"
        android:layout_width="wrap_content"
        android:gravity="center"
        android:hint="Стоимость" />
    <EditText
        android:id="@+id/edit_main_amount"
        android:layout_width="wrap_content"
        android:inputType="number"
        android:gravity="center"
        android:hint="Количество" />
</TableRow>

```

Рисунок 40 – Поля для ввода в файле разметки activity\_main.xml

```

<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="center">
    <Button
        android:id="@+id/button_main_add"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Добавить"/>
    <Button
        android:id="@+id/button_main_get"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Получить"/>
    <Button
        android:id="@+id/button_main_delete"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Удалить"/>
</LinearLayout>
<Button
    android:id="@+id/button_main_set"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Установить значение по ID"/>

```

Рисунок 41 – Кнопки в файле разметки activity\_main.xml

```

<TableRow
    android:layout_width="match_parent"
    android:layout_height="50dp"
    android:gravity="center"
    android:showDividers="middle">
    <TextView
        android:id="@+id/text_main_id"
        android:layout_width="0dp"
        android:layout_weight="1"
        android:gravity="center"
        android:textSize="14sp" />
    <TextView
        android:id="@+id/text_main_name"
        android:layout_width="0dp"
        android:layout_weight="3"
        android:gravity="center"
        android:textSize="14sp" />
    <TextView
        android:id="@+id/text_main_weight"
        android:layout_width="0dp"
        android:layout_weight="2"
        android:gravity="center"
        android:textSize="14sp" />
    <TextView
        android:id="@+id/text_main_price"
        android:layout_width="0dp"
        android:layout_weight="3"
        android:gravity="center"
        android:textSize="14sp" />
    <TextView
        android:id="@+id/text_main_amount"
        android:layout_width="0dp"
        android:layout_weight="3"
        android:gravity="center"
        android:textSize="14sp" />
</TableRow>

```

Рисунок 42 – Текстовые поля в файле разметки activity\_main.xml

## 2.2.2 Реализация

Сначала создадим класса DBEntry, в котором будет храниться информация строки таблицы. Для выполнения данной работы элементом таблицы был выбран товар, для которого были выбраны следующие поля:

- Целочисленное поле id;
- Строковое поле name;
- Строковое поле weight;
- Строковое поле price;
- Целочисленное поле amount.

Каждое поле соответствует одноименному полю класса DBEntry. Также в классе присутствует конструктор и методы доступа к полям (Рисунок 43).

```
public class DBEntry
{
    private final int id;
    private final String name;
    private final String weight;
    private final String price;
    private final int amount;

    public DBEntry(int id, String name, String weight, String price, int amount)
    {
        this.id = id;
        this.name = name;
        this.weight = weight;
        this.price = price;
        this.amount = amount;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getWeight() {
        return weight;
    }

    public String getPrice() {
        return price;
    }

    public int getAmount() {
        return amount;
    }
}
```

**Рисунок 43 – Описание класса DBEntry**

Дальше создадим класс DBHelper, наследуемый от класса SQLiteOpenHelper, предназначенный для работы с базой данных из класса MainActivity.

На рисунке 44 представлены константы класса DBHelper, а также конструктор и методы onCreate() и onUpgrade(), которые необходимо реализовать при наследовании от класса SQLiteOpenHelper.

```

public class DBHelper extends SQLiteOpenHelper
{
    private static final String DATABASE_NAME = "database.db";
    private static final int DATABASE_VERSION = 2;
    private static final String TABLE_NAME = "products";
    private static final String COLUMN_ID = "id";
    private static final String COLUMN_NAME = "name";
    private static final String COLUMN_WEIGHT = "weight";
    private static final String COLUMN_PRICE = "price";
    private static final String COLUMN_AMOUNT = "amount";

    private static final String CREATE_TABLE =
        "CREATE TABLE " + TABLE_NAME + " (" +
            COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
            COLUMN_NAME + " TEXT, " +
            COLUMN_WEIGHT + " TEXT, " +
            COLUMN_PRICE + " TEXT, " +
            COLUMN_AMOUNT + " INTEGER);";

    public DBHelper(@Nullable Context context)
    {
        super(context, DATABASE_NAME, factory: null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db)
    {
        db.execSQL(CREATE_TABLE);
    }

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

```

**Рисунок 44 – Описание обязательных методов класса DBHelper**

В дополнение к обязательным методам также были реализованы методы для работы с базой данных:

- addEntry() для добавление строки в таблицу;
- getEntry() для получения строки таблицы по id;
- deleteEntry() для удаления строки из таблицы по id;
- setEntry() для установления значений строки таблицы.

Реализация данных методов представлена на рисунках 45-48.

```

public boolean addEntry(DBEntry entry)
{
    SQLiteDatabase db = getWritableDatabase();
    ContentValues values = new ContentValues();
    values.put(COLUMN_NAME, entry.getName());
    values.put(COLUMN_WEIGHT, entry.getWeight());
    values.put(COLUMN_PRICE, entry.getPrice());
    values.put(COLUMN_AMOUNT, entry.getAmount());
    long result = db.insert(TABLE_NAME, nullColumnHack: null, values);
    db.close();
    return result != -1;
}

```

**Рисунок 45 – Описание метода addEntry() класса DBHelper**

```

public DBEntry getEntry(String id)
{
    SQLiteDatabase db = getReadableDatabase();
    Cursor cursor = db.query(
        TABLE_NAME,
        new String[]{COLUMN_ID, COLUMN_NAME, COLUMN_WEIGHT, COLUMN_PRICE, COLUMN_AMOUNT},
        selection: COLUMN_ID + " = " + id,
        selectionArgs: null,
        groupBy: null,
        having: null,
        orderBy: null);
    DBEntry entry = null;
    if (cursor.moveToFirst())
    {
        entry = new DBEntry(cursor.getInt( columnIndex: 0),
            cursor.getString( columnIndex: 1),
            cursor.getString( columnIndex: 2),
            cursor.getString( columnIndex: 3),
            cursor.getInt( columnIndex: 4));
    }
    cursor.close();
    db.close();
    return entry;
}

```

**Рисунок 46 – Описание метода getEntry() класса DBHelper**

```

public boolean deleteEntry(String id)
{
    SQLiteDatabase db = getWritableDatabase();
    int result = db.delete(TABLE_NAME,
        whereClause: COLUMN_ID + " = " + id,
        whereArgs: null);
    db.close();
    return result > 0;
}

```

**Рисунок 47 – Описание метода deleteEntry() класса DBHelper**

```

public boolean setEntry(DBEntry newEntry)
{
    SQLiteDatabase db = getWritableDatabase();
    ContentValues values = new ContentValues();
    values.put(COLUMN_ID, newEntry.getId());
    values.put(COLUMN_NAME, newEntry.getName());
    values.put(COLUMN_WEIGHT, newEntry.getWeight());
    values.put(COLUMN_PRICE, newEntry.getPrice());
    values.put(COLUMN_AMOUNT, newEntry.getAmount());
    long result = db.update(TABLE_NAME, values,
        whereClause: COLUMN_ID + " = " + newEntry.getId(),
        whereArgs: null);
    db.close();
    return result > 0;
}

```

**Рисунок 48 – Описание метода setEntry() класса DBHelper**

Далее в методе onCreate() класса MainActivity была реализована привязка нажатия на кнопки к выполнению соответствующего запроса с помощью класса DBHelper.

На рисунке 49 представлено получение всех элементов разметки.

```

@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    EdgeToEdge.enable( $this$enableEdgeToEdge: this);
    setContentView(R.layout.activity_main);
    ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main), (v, insets) ->
    {
        Insets systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars());
        v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom);
        return insets;
    });

    EditText editId = findViewById(R.id.edit_main_id);
    EditText editName = findViewById(R.id.edit_main_name);
    EditText editWeight = findViewById(R.id.edit_main_weight);
    EditText editPrice = findViewById(R.id.edit_main_price);
    EditText editAmount = findViewById(R.id.edit_main_amount);

    TextView textId = findViewById(R.id.text_main_id);
    TextView textName = findViewById(R.id.text_main_name);
    TextView textWeight = findViewById(R.id.text_main_weight);
    TextView textPrice = findViewById(R.id.text_main_price);
    TextView textAmount = findViewById(R.id.text_main_amount);

    DBHelper helper = new DBHelper( context: this);

```

**Рисунок 49 – Описание метода onCreate() класса MainActivity, часть 1**

На рисунке 50 представлена реализация функционала кнопок «Добавить» и «Получить».

```

Button buttonAdd = findViewById(R.id.button_main_add);
buttonAdd.setOnClickListener(v ->
{
    DBEntry entry = new DBEntry(
        id: 0,
        editName.getText().toString(),
        editWeight.getText().toString(),
        editPrice.getText().toString(),
        Integer.parseInt(editAmount.getText().toString()));
    boolean result = helper.addEntry(entry);
    if (!result)
    {
        Toast.makeText( context: this, text: "Произошла ошибка", Toast.LENGTH_SHORT).show();
    }
});

Button buttonGet = findViewById(R.id.button_main_get);
buttonGet.setOnClickListener(v ->
{
    DBEntry entry = helper.getEntry(editId.getText().toString());
    if (entry == null)
    {
        Toast.makeText( context: this, text: "Запись не найдена", Toast.LENGTH_SHORT).show();
        return;
    }
    textId.setText(String.valueOf(entry.getId()));
    textName.setText(entry.getName());
    textWeight.setText(entry.getWeight());
    textPrice.setText(entry.getPrice());
    textAmount.setText(String.valueOf(entry.getAmount()));
});

```

**Рисунок 50 – Описание метода onCreate() класса MainActivity, часть 2**

На рисунке 51 представлена реализация функционала кнопок «Удалить» и «Установить строку по ID»



```

Button buttonDelete = findViewById(R.id.button_main_delete);
buttonDelete.setOnClickListener(v ->
{
    boolean result = helper.deleteEntry(editId.getText().toString());
    if (!result)
    {
        Toast.makeText(context: this, text: "Запись не найдена", Toast.LENGTH_SHORT).show();
    }
});

Button buttonSet = findViewById(R.id.button_main_set);
buttonSet.setOnClickListener(v ->
{
    DBEntry entry = new DBEntry(
        Integer.parseInt(editId.getText().toString()),
        editName.getText().toString(),
        editWeight.getText().toString(),
        editPrice.getText().toString(),
        Integer.parseInt(editAmount.getText().toString()));
    boolean result = helper.setEntry(entry);
    if (!result)
    {
        Toast.makeText(context: this, text: "Запись не найдена", Toast.LENGTH_SHORT).show();
    }
});
}

```

Рисунок 51 – Описание метода onCreate() класса MainActivity, часть 3

### 2.2.3 Тестирование

Откроем приложение (Рисунок 52).

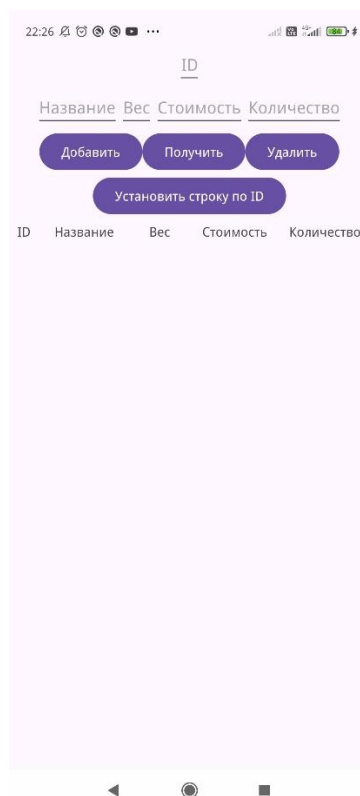
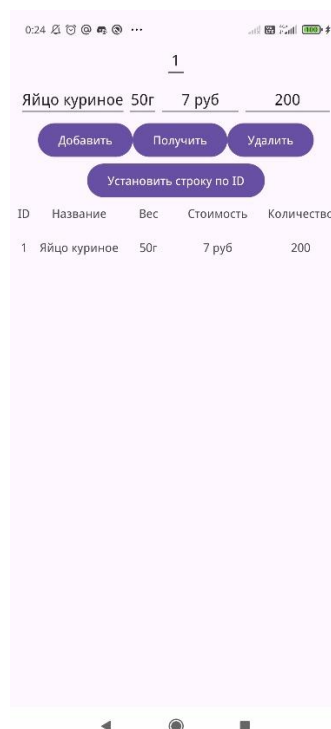


Рисунок 52 – Начальный экран приложения по работе с базой данных

Введем данные строки и затем нажмем кнопки «Добавить» и «Получить» (Рисунок 53).



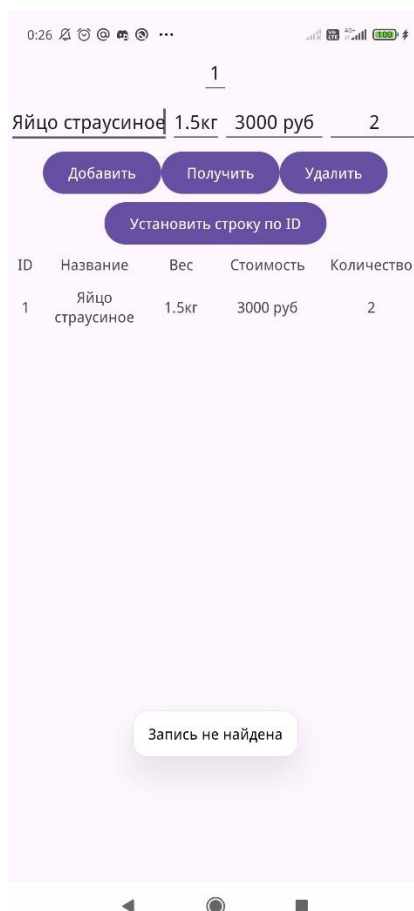
**Рисунок 53 – Добавление строки в таблицу**

Введем данные строки повторно, затем нажмем кнопки «Установить строку по ID» и «Получить» (Рисунок 54).



**Рисунок 54 – Изменение строки таблицы по ID**

Нажмем кнопки «Удалить» и «Получить». В результате строка по ID 1 не была найдена, как и ожидалось (Рисунок 55).



**Рисунок 55 – Удаление строки таблицы по ID**  
Приложение работает успешно.

## **ЗАКЛЮЧЕНИЕ**

В ходе работы были получены знания по работе с SharedPreferences, а также базами данных в Android Studio. Полученные знания были закреплены путём сохранения имени пользователя в SharedPreferences приложения, а также создания базы данных и работы с ней при помощи вспомогательного класса SQLiteOpenHelper.