



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 «Проектирование и разработка мобильных приложений»

**Выполнил:**

Студент группы ИКБО-21-23

Лисовский И.В.

**Проверил:**

Старший преподаватель кафедры  
МОСИТ

Шешуков Л.С.

Москва 2025 г.

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ .....	3
1 ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ.....	4
1.1 Класс SharedPreferences .....	4
1.2 Основные понятия баз данных.....	9
1.3 Основные команды языка SQL .....	11
1.4 Работа с СУБД в Android .....	18
2 ПРАКТИЧЕСКАЯ ЧАСТЬ .....	35
2.1 Реализация записи, получения, изменения и удаления имени пользователя приложения через SharedPreferences.....	35
2.2 Создание тематической базы данных. Реализация методов записи, поиска, изменения и удаления данных .....	37
ЗАКЛЮЧЕНИЕ.....	45

## **ВВЕДЕНИЕ**

В данной практической работе были реализованы две функциональные задачи, направленные на закрепление навыков работы с локальным хранилищем данных в Android-приложениях на языке Java.

Первая задача заключалась в использовании механизма `SharedPreferences` для сохранения, получения, изменения и удаления имени пользователя. Этот подход является простым и эффективным решением для хранения небольших объёмов данных, таких как настройки или персонализированная информация.

Вторая задача была посвящена созданию и управлению локальной базой данных `SQLite`, где на примере автосалона реализована таблица с информацией об автомобилях. Для работы с базой данных были созданы методы для добавления, отображения, обновления и удаления записей. Все взаимодействия с базой данных происходят через класс, унаследованный от `SQLiteOpenHelper`.

# 1 ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ

## 1.1 Класс SharedPreference

В предыдущей практике мы использовали файловую систему для сохранения данных, что удобно при передаче информации между активностями и даже приложениями. Однако для хранения простых данных — таких как строки, числа или логические значения — использование отдельных файлов не всегда оправдано. В таких случаях гораздо эффективнее применять механизм SharedPreferences.

SharedPreferences — это встроенный инструмент Android для хранения простых данных в формате «ключ-значение». Он отлично подходит для сохранения настроек приложения, авторизационной информации и других небольших данных, которые должны быть доступны между сессиями работы приложения.

Где используется SharedPreferences:

- настройки пользователя: например, выбранная тема, язык интерфейса или другие предпочтения,
- состояние приложения: можно сохранить, какую вкладку пользователь открыл последней или что он ввёл в форму, чтобы при следующем запуске восстановить то же состояние,
- небольшие объёмы данных: SharedPreferences избавляет от необходимости подключать базу данных, если нужно хранить лишь немного информации.

Пример сохранения данных показан на рисунке 1.

```

// Получаем доступ к файлу настроек "myPreferences"
SharedPreferences sharedPreferences = getSharedPreferences("myPreferences", MODE_PRIVATE);

// Создаём редактор для записи данных
SharedPreferences.Editor editor = sharedPreferences.edit();

// Сохраняем данные
editor.putString("username", "User123");
editor.putInt("sessionCount", 5);
editor.putBoolean("loggedIn", true);

// Применяем изменения
editor.apply();

```

Рисунок 1 – Сохранение данных

Метод `getSharedPreferences()` принимает два параметра: имя файла настроек (в данном случае `"myPreferences"`) и режим доступа (`MODE_PRIVATE`, при котором настройки доступны только вашему приложению). Если файл настроек ещё не существует, он будет создан автоматически.

Метод `edit()` возвращает объект `SharedPreferences.Editor`, с помощью которого можно добавлять или изменять данные.

Чтобы получить ранее сохранённые значения, сначала нужно получить доступ к объекту `SharedPreferences`. Это показано на рисунке 2.

```

SharedPreferences sharedPreferences = getSharedPreferences("myPreferences", MODE_PRIVATE);

// Извлечение данных
String username = sharedPreferences.getString("username", "defaultUsername");
int sessionCount = sharedPreferences.getInt("sessionCount", 0);
boolean isLoggedIn = sharedPreferences.getBoolean("loggedIn", false);

```

Рисунок 2 – Чтение данных

Метод `getString(key, defaultValue)` возвращает строковое значение, связанное с указанным ключом. Если такого ключа нет в настройках, будет возвращено значение по умолчанию, переданное вторым параметром.

Аналогичным образом работают методы `getInt()` и `getBoolean()` для получения чисел и логических значений соответственно.

Этот подход позволяет безопасно извлекать данные без риска возникновения ошибок в случае отсутствия нужного ключа.

На рисунке 3 показан пример работы программы.

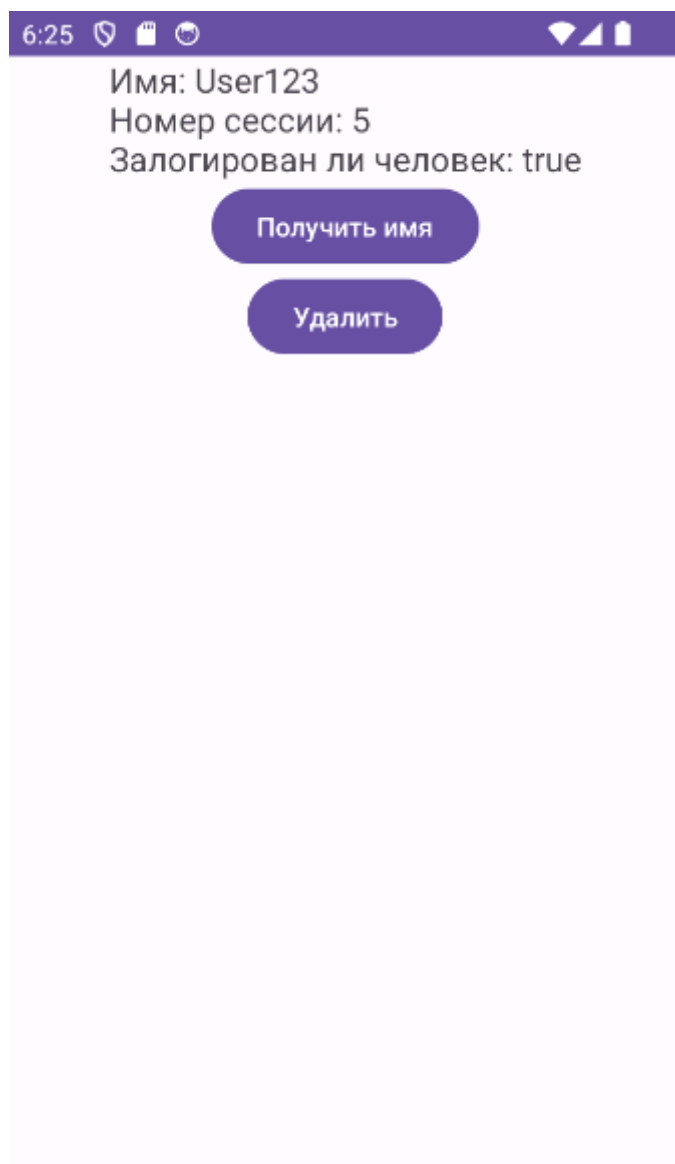


Рисунок 3 – Работоспособность программы

Чтобы удалить определённые значения из настроек или полностью очистить все сохранённые данные, используется редактор `SharedPreferences.Editor`. Пример показан на рисунке 4.

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

// Удаляем значение по ключу "username"
editor.remove("username");

// Полностью очищаем все данные из настроек
editor.clear();

// Применяем изменения
editor.apply();
```

Рисунок 4 – Удаление данных

- метод `remove("ключ")` удаляет конкретное значение, связанное с заданным ключом,
- метод `clear()` удаляет все данные, хранящиеся в `SharedPreferences`,
- метод `apply()` сохраняет изменения асинхронно.

Это удобно, например, при выходе пользователя из приложения или при сбросе настроек к значениям по умолчанию.

Удаление данных продемонстрировано на рисунке 5.

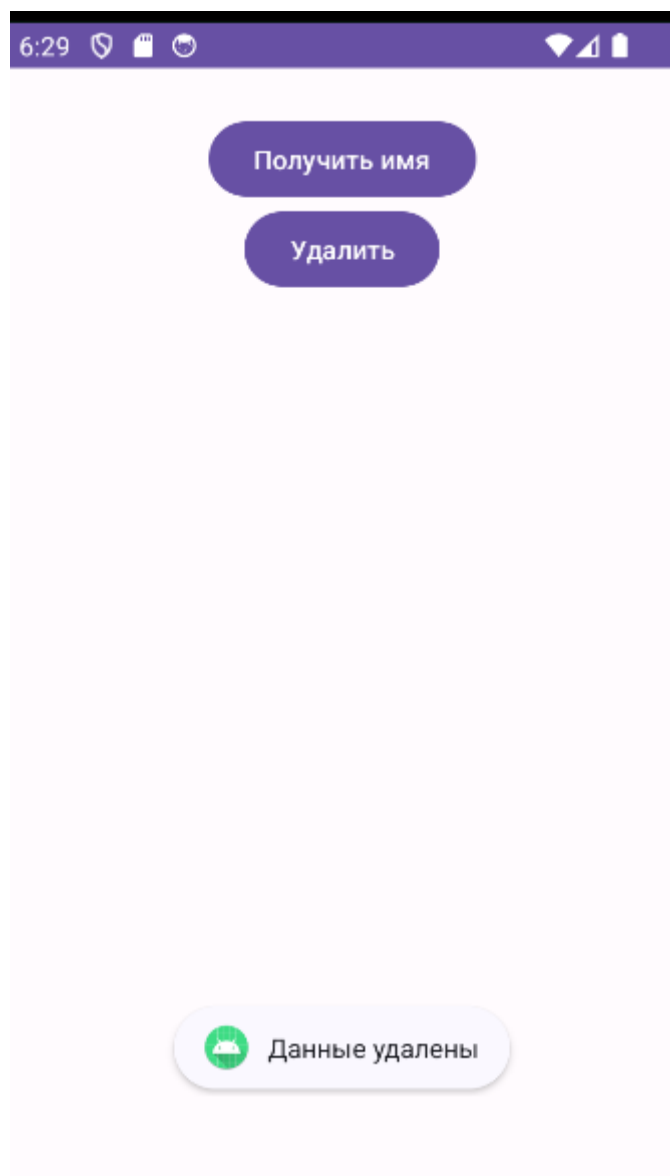


Рисунок 5 – Работоспособность программы

Если попробовать получить данные после того, как они были удалены, то получим данные, прописанные по умолчанию. Это показано на рисунке 6.



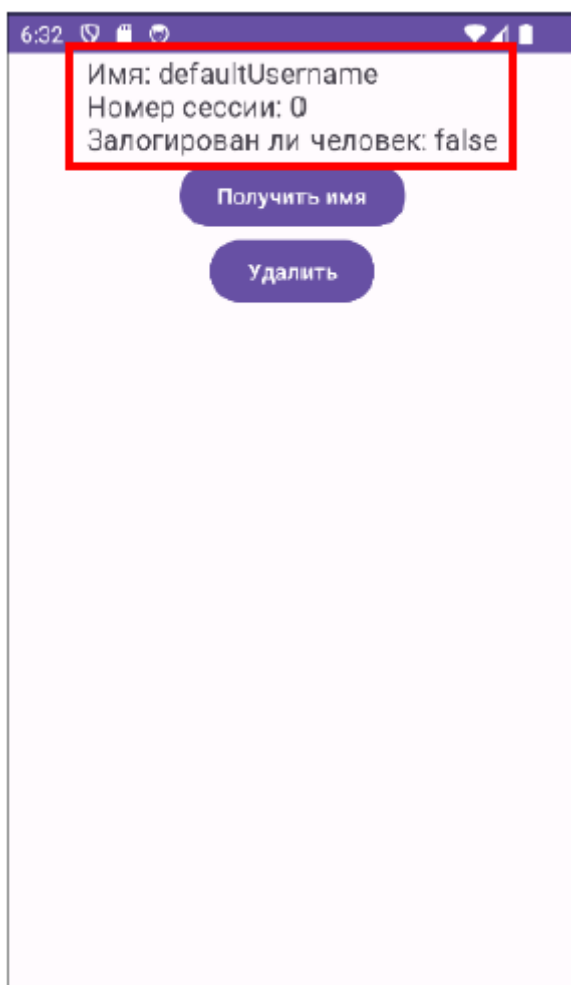


Рисунок 6 – Получение дефолтных значений

## 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.

Это выглядит примерно как Excel таблица. Есть колонки с заголовками, и информация внутри. На рисунке 7 показан пример интерфейса таблицы.

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 7 – Пример интерфейса таблицы

Это одна из наиболее распространённых моделей организации данных, которая обеспечивает удобное управление и быстрый доступ к информации. Ниже приведены ключевые элементы данной концепции:

1) таблицы - данные размещаются в таблицах, каждая из которых соответствует определённой сущности (например, клиенты, товары, заказы),

2) строки и столбцы - каждая строка представляет собой отдельный объект (экземпляр сущности), а столбцы содержат его характеристики. К примеру, в таблице с клиентами строки будут отражать конкретных клиентов, а столбцы — такие данные, как имя, адрес и номер телефона,

3) первичные ключи (Primary Keys) - для уникальной идентификации каждой строки используется первичный ключ — уникальное значение, которое однозначно определяет запись. Обычно такие ключи генерируются автоматически, чтобы избежать дублирования,

4) внешние ключи (Foreign Keys) - обеспечивают связь между таблицами. Например, в таблице заказов внешний ключ может ссылаться на таблицу клиентов, указывая, кто именно сделал заказ,

5) нормализация - чтобы минимизировать избыточность и предотвратить ошибки, данные проходят нормализацию — процесс разделения информации по нескольким связанным таблицам. Это упрощает сопровождение и обновление данных,

6) индексы - создаются для ускорения поиска и фильтрации данных. Они могут быть заданы для одного или нескольких столбцов, позволяя

системе быстро находить нужные записи.

Эта структура лежит в основе реляционных баз данных, в которых для работы с данными применяется язык SQL (Structured Query Language — язык структурированных запросов).

### 1.3 Основные команды языка SQL

SQL (Structured Query Language) — это язык, с помощью которого осуществляется взаимодействие с базой данных. Прежде чем выполнять запросы, необходимо создать саму базу и таблицы в ней.

Для создания таблицы используется команда CREATE TABLE. Её синтаксис показан на рисунке 8.

```
CREATE TABLE имя_таблицы (  
    имя_столбца1 тип_данных [ограничения],  
    имя_столбца2 тип_данных [ограничения],  
    ...  
);
```

Рисунок 8 – Создание таблицы

Имя таблицы должно быть уникальным в рамках базы данных и не должно начинаться с `sqlite_`, так как такие имена зарезервированы системой SQLite для служебных нужд.

Внутри скобок указываются имена столбцов, их типы данных и, при необходимости, ограничения или дополнительные атрибуты.

Создадим таблицу `student` для хранения информации о студентах — их ФИО, возрасте и поле и покажем это на рисунке 9.

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

Рисунок 9 – Создание таблицы student

id INTEGER PRIMARY KEY AUTOINCREMENT — создаёт уникальный идентификатор, который автоматически увеличивается с добавлением каждой новой записи.

Если попытаться повторно выполнить такую команду, возникнет ошибка, потому что таблица с таким именем уже существует. Чтобы этого избежать, можно использовать конструкцию IF NOT EXISTS, которая позволяет создать таблицу только если она ещё не была создана, что показано на рисунке 10.

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

Рисунок 10 – Использование конструкции IF NOT EXISTS

Для вставки записей в таблицу используется команда INSERT. Общий синтаксис показан на рисунке 11.

```
INSERT INTO имя_таблицы (столбец1, столбец2, ..., столбецN)  
VALUES (значение1, значение2, ..., значениеN);
```

Рисунок 11 – Добавление данных

После имени таблицы указываются нужные столбцы, а затем в VALUES — соответствующие значения. Значения передаются строго в том же порядке, что и указанные столбцы.

На рисунке 12 покажем вставку данных на примере.

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

Рисунок 12 – Пример вставки данных

В данном случае:

- fio получит значение 'Муравьёва Екатерина Андреевна',
- age — 25,
- gender — 'женский'.

После выполнения этой команды новая запись появится в таблице student.

Важно понимать, что при добавлении записей в таблицу необязательно указывать значения для всех столбцов. Например, в приведённом ранее примере мы не задавали значение для поля id, поскольку оно формируется автоматически благодаря атрибуту AUTOINCREMENT.

Можно также выполнять вставку без явного перечисления столбцов, как на рисунке 13.

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

Рисунок 13 – Вставка без явного перечисления столбцов

Однако в таком случае необходимо обязательно указать значения для всех столбцов, в том числе и для id, и при этом соблюдать точный порядок следования столбцов, как он определён при создании таблицы.

Теперь в таблице student появилась первая запись. Но для полноценной работы с данными этого недостаточно. Добавим ещё несколько студентов, чтобы впоследствии можно было выполнять выборку, фильтрацию и другие

SQL-операции. На рисунке 14 показана новая таблица.

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 14 – Обновленная база данных

Для получения данных из базы данных SQLite используется команда SELECT. В простейшем виде её синтаксис показан на рисунке 15.

```
SELECT список_столбцов FROM имя_таблицы;
```

Рисунок 15 – Получение данных в общем виде

Часто нам нужно извлекать из базы только те записи, которые соответствуют определённому условию. Для этого в команде SELECT используется оператор WHERE, за которым следует само условие. Это показано на рисунке 16.

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

Рисунок 16 – Использование оператора WHERE

- SELECT — указывает, какие столбцы мы хотим получить,
- FROM — из какой таблицы берутся данные,
- WHERE — какое условие должны удовлетворять строки.

Например, предположим, что мы хотим получить информацию обо всех студентах, которым 24 года. Формируем запрос по смыслу: «получить все записи о студентах, у которых возраст равен 24». В SQL это будет выглядеть как показано на рисунке 17.

```
SELECT * FROM student WHERE age = 24;
```

Рисунок 17 – Получение конкретных данных

Символ \* означает, что мы выбираем все столбцы. Конечно, при необходимости мы можем указать конкретные столбцы вместо звёздочки, если нужны не все данные.

Если бы у нас не было базы данных, а были обычные Excel-файлы, мы бы выполнили то же самое следующим образом:

- 1) Открыли бы файл с нужными данными (например, student).
- 2) Установили бы фильтр на колонку «Возраст» со значением 24.

То есть, независимо от инструмента — будь то база данных или электронная таблица — мы должны знать, где хранятся данные (название таблицы или файла) и по какому столбцу мы хотим их отфильтровать. Это не какая-то уникальная особенность баз данных — аналогичные действия можно выполнять и в обычном Excel.

Для изменения уже существующих записей в SQLite применяется команда UPDATE. Чтобы обновить только нужные строки, мы можем уточнить условие с помощью оператора WHERE. Синтаксис показан на рисунке 18.

```
UPDATE имя_таблицы  
SET столбец1 = новое_значение1, столбец2 = новое_значение2  
WHERE условие;
```

Рисунок 18 – Обновление данных

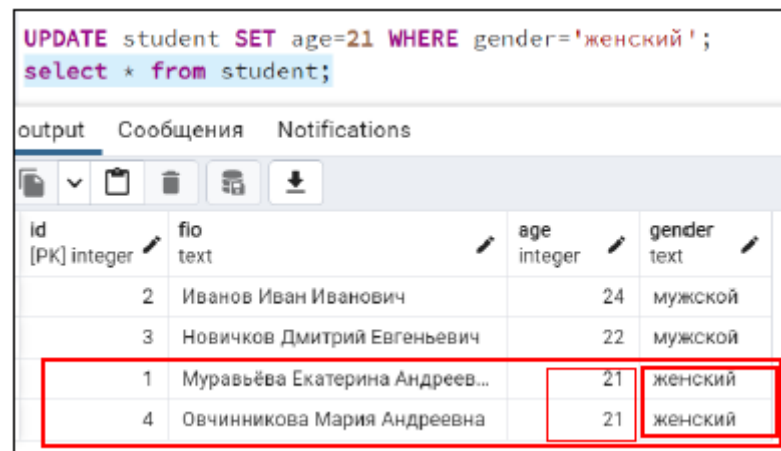
Предположим, что мы хотим изменить данные в таблице student: всем студенткам (то есть тем, у кого пол — "женский") мы устанавливаем возраст 21. Покажем это на рисунке 19.

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

Рисунок 19 – Обновление данных на примере

Таким образом, мы обновляем только те записи, которые соответствуют заданному условию, не затрагивая остальные.

Обновленную базу данных покажем на рисунке 20.



id	fio	age	gender
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 20 – Обновленная база данных

Команда DELETE используется для удаления данных из базы. Её основной синтаксис показан на рисунке 21.

```
DELETE FROM имя_таблицы WHERE условие;
```

Рисунок 21 – Синтаксис команды DELETE

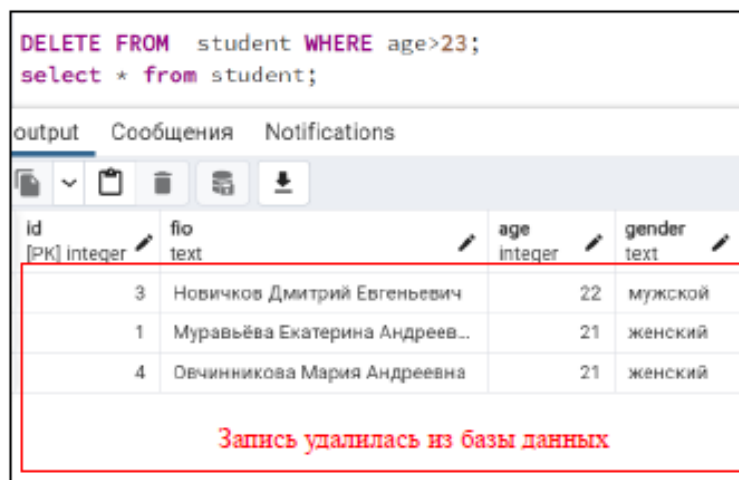
Допустим, мы хотим удалить из таблицы student всех студентов, которым больше 23 лет. Тогда наш запрос будет выглядеть как показано на рисунке 22.

```
DELETE FROM student WHERE age > 23;
```



Рисунок 22 – Удаление конкретных данных

Обновленную базу данных покажем на рисунке 23.



The screenshot shows a database management interface. At the top, there is a text area with the following SQL commands: `DELETE FROM student WHERE age>23;` and `select * from student;`. Below the text area are tabs labeled 'output', 'Сообщения', and 'Notifications'. Under the 'output' tab, there is a toolbar with icons for copy, paste, delete, and download. Below the toolbar is a table with the following columns: 'id' (integer, primary key), 'fio' (text), 'age' (integer), and 'gender' (text). The table contains three rows of data. A red box highlights the entire table content, and a red message at the bottom states 'Запись удалась из базы данных'.

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

Запись удалась из базы данных

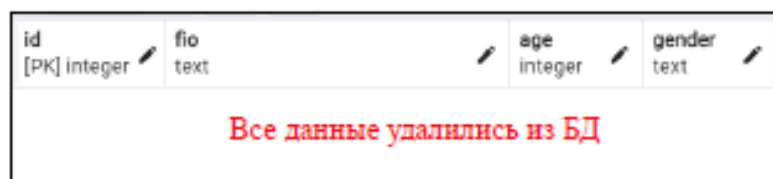
Рисунок 23 – Обновленная база данных

Если мы хотим удалить абсолютно все записи из таблицы, не задавая условий, можно опустить WHERE. Покажем синтаксис на рисунке 24.

```
DELETE FROM student;
```

Рисунок 24 – Удаление всех записей

Результат такого запроса покажем на рисунке 25.



The screenshot shows the same database management interface as in Figure 23. The table is now empty. A red message at the bottom states 'Все данные удалились из БД'.

id	fio	age	gender
[PK] integer	text	integer	text

Все данные удалились из БД

Рисунок 25 – Обновленная база данных

Чтобы удалить саму таблицу вместе со всеми её данными, используется команда DROP TABLE. Синтаксис такого запроса показан на рисунке 26.

```
DROP TABLE имя_таблицы;
```

Рисунок 26 – Удаление таблицы

Результат работы такого запроса покажем на рисунке 27.

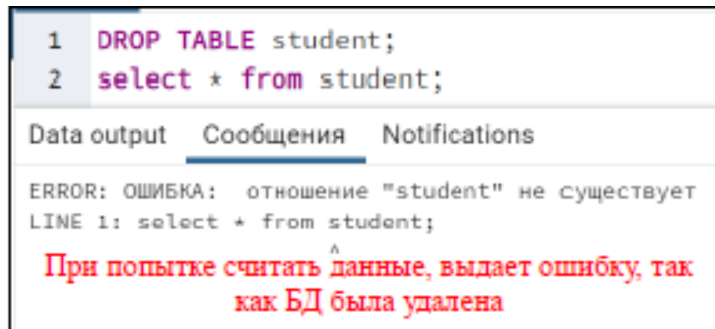


Рисунок 27 – Результат удаления базы данных

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. Покажем это на рисунке 28.



Рисунок 28 – Использование IF EXISTS при удалении

## 1.4 Работа с СУБД в Android

В Android-разработке для управления базами данных мы часто используем SQLite — встроенную, лёгкую систему управления базами данных (СУБД), которая поддерживает большую часть возможностей SQL.

Для взаимодействия с базой через SQLite мы можем использовать интерфейс Cursor, который обеспечивает произвольный доступ к результатам SQL-запросов.

Основные средства работы с базами данных предоставляет пакет `android.database`, а работа с `SQLite` организована через `android.database.sqlite`.

- класс `SQLiteDatabase` представляет саму базу данных и позволяет выполнять с ней различные операции: чтение, запись, обновление и удаление данных,
- класс `SQLiteCursor` используется для выполнения запросов и получения набора строк, соответствующих этим запросам,
- `SQLiteQueryBuilder` помогает нам формировать SQL-запросы программно,
- SQL-выражения представлены с помощью `SQLiteStatement`, что позволяет вставлять в запросы значения динамически, используя плейсхолдеры,
- класс `SQLiteOpenHelper` используется для создания и обновления базы данных. С его помощью мы можем создавать базу и все необходимые таблицы, если они ещё не существуют.

Если мы попытаемся прочесть данные из базы, которая была ранее удалена, — произойдёт ошибка, так как файл базы данных больше не существует.

В качестве примера давайте создадим приложение для хранения контактов. У нас будет одна сущность — контакт, включающая имя, номер телефона и уникальный идентификатор (первичный ключ).

Сначала создадим Java-класс для представления контактов и покажем его на рисунке 29.

```

public class Contact {
    private int id;          // идентификатор
    private String name;     // имя
    private String phone;    // номер телефона

    public Contact(int id, String name, String phone) {
        this.id = id;
        this.name = name;
        this.phone = phone;
    }

    // Геттеры и сеттеры
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getPhone() { return phone; }
    public void setPhone(String phone) { this.phone = phone; }
}

```

Рисунок 29 – Созданный Java класс

Далее мы создаём отдельный класс-помощник DatabaseHelper для работы с базой данных. Этот класс должен наследовать SQLiteOpenHelper и переопределять как минимум два метода: onCreate() и onUpgrade(). В этом классе будет реализована логика создания и обновления базы.

Метод onCreate(). Этот метод вызывается, когда мы впервые пытаемся получить доступ к базе данных, но она ещё не создана. Внутри него мы формируем SQL-запрос для создания таблицы и выполняем его с помощью execSQL(). Покажем код этого метода на рисунке 30.

```

@Override
public void onCreate(SQLiteDatabase db) {
    String createTable = "CREATE TABLE " + TABLE_NAME + " ("
        + COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, "
        + COLUMN_NAME + " TEXT, "
        + COLUMN_PHONE + " TEXT)";
    db.execSQL(createTable);
}

```

Рисунок 30 – Метод onCreate()

Метод onUpgrade() вызывается, если требуется обновить структуру базы данных, например, при изменении её версии. В простейшем случае мы можем удалить существующую таблицу и создать новую. Покажем код этого метода на рисунке 31.

```

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

```

Рисунок 31 – Метод onUpgrade()

Здесь используется команда DROP TABLE, чтобы удалить существующую таблицу перед её пересозданием.

Стоит отметить, что такой способ обновления подходит только для примеров или тестов. В реальных проектах, когда важно сохранить существующие данные, метод onUpgrade() должен содержать более сложную логику: добавление новых столбцов, удаление устаревших, миграция данных и т. д.

Теперь, когда мы создали базу данных и таблицу, следующим шагом будет её наполнение данными.

Кстати, созданную базу можно найти и открыть в Device File Explorer по пути /data/data/имя\_вашего\_пакета/databases.

Для выполнения операций добавления, изменения и удаления данных в

SQLite мы используем методы `insert()`, `update()` и `delete()` класса `SQLiteDatabase`.

Чтобы получить доступ к базе данных, мы можем воспользоваться методами `getReadableDatabase()` (для чтения) или `getWritableDatabase()` (для записи). Так как в этом случае мы планируем добавлять данные, используем второй вариант. Покажем на рисунке 32 код для добавления нового контакта.

```
// Добавление нового контакта
public boolean addContact(Contact contact) {
    SQLiteDatabase db = this.getWritableDatabase();
    ContentValues cv = new ContentValues();
    cv.put(COLUMN_NAME, contact.getName());
    cv.put(COLUMN_PHONE, contact.getPhone());
    long result = db.insert(TABLE_NAME, null, cv);
    db.close();
    return result != -1;
}
```

Рисунок 32 – Добавление нового контакта

Для добавления или обновления данных мы создаём объект `ContentValues`, который представляет собой словарь с парами «ключ–значение». Метод `put()` позволяет нам добавить данные в этот словарь: первый параметр — это название столбца (ключ), второй — соответствующее значение.

Метод `insert()` принимает три параметра:

- имя таблицы,
- `nullColumnHack` — имя столбца, в который можно вставить `NULL`, если остальные значения отсутствуют (часто передаётся как `null`),
- объект `ContentValues` с данными для добавления.

После завершения работы с базой данных мы закрываем соединение с помощью `close()`.

Теперь, на рисунке 33 покажем удаление записи.

```
// Удаление контакта по номеру телефона  
public boolean deleteContact(String phone) {  
    SQLiteDatabase db = this.getWritableDatabase();  
    int result = db.delete(TABLE_NAME, COLUMN_PHONE + " = ?", new String[]{phone});  
    db.close();  
    return result > 0;  
}
```

Рисунок 33 – Удаление записи

Метод delete() принимает:

- имя таблицы;
- условие WHERE, в котором ? — это плейсхолдер;
- массив значений, которые будут подставлены вместо ?.

Такой подход позволяет безопасно передавать параметры и защищает от SQL-инъекций.

На рисунке 34 покажем метод поиска записи.

```

// Поиск контакта по номеру телефона
public Contact findContact(String phone) {
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = db.query(
        TABLE_NAME,
        new String[]{COLUMN_ID, COLUMN_NAME, COLUMN_PHONE},
        COLUMN_PHONE + " = ?",
        new String[]{phone},
        null, null, null
    );

    if (cursor != null && cursor.moveToFirst()) {
        Contact contact = new Contact(
            cursor.getInt(0),
            cursor.getString(1),
            cursor.getString(2)
        );
        cursor.close();
        db.close();
        return contact;
    }

    if (cursor != null) {
        cursor.close();
    }
    db.close();
    return null;
}

```

Рисунок 34 – Метод удаления записи

Метод `query()` позволяет нам выполнять выборку по заданному условию. Если запись найдена, мы создаём объект `Contact` на основе данных из курсора и возвращаем его. В конце не забываем закрыть `Cursor` и базу данных.

Когда мы ищем контакт по номеру телефона, нам не нужно вносить изменения в базу данных — достаточно использовать метод `getReadableDatabase()`, который предоставляет доступ к данным только для чтения.

Для управления результатами выборки мы используем класс `Cursor`,



который предоставляет набор методов:

- moveToFirst() — перемещает курсор к первой строке выборки;
- moveToNext() — переходит к следующей строке;
- методы get\*() (например, getString(), getInt(), getLong()) позволяют

получить данные из определённого столбца по его индексу в текущей строке.

Когда работа с курсором завершена, мы обязательно закрываем его с помощью close(), чтобы освободить ресурсы.

На рисунке 35 покажем метод для получения всех записей из базы.

```
public List<Contact> getAllContacts() {
    List<Contact> contactList = new ArrayList<>();
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = db.rawQuery("SELECT * FROM " + TABLE_NAME, null);

    if (cursor.moveToFirst()) {
        do {
            Contact contact = new Contact(
                cursor.getInt(0),
                cursor.getString(1),
                cursor.getString(2)
            );
            contactList.add(contact);
        } while (cursor.moveToNext());
    }

    cursor.close();
    db.close();
    return contactList;
}
```

Рисунок 35 – Получение всех записей

Метод rawQuery() выполняет произвольный SQL-запрос и возвращает Cursor с результатами. Сначала мы проверяем, удалось ли получить хотя бы одну строку — если moveToFirst() вернёт false, значит в базе данных нет данных. Далее, используя moveToNext(), мы проходим по всем строкам результата, создавая объекты Contact и добавляя их в список.

На рисунке 36 покажем метод обновления данных.

```

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);

    int result = db.update(
        TABLE_NAME,
        cv,
        COLUMN_PHONE + " = ?",
        new String[]{oldPhone}
    );

    db.close();
    return result > 0;
}

```

Рисунок 36 – Метод обновления данных

Здесь мы обновляем строку, в которой номер телефона совпадает со старым значением (oldPhone), заменяя имя и номер на новые значения. Возвращаем true, если обновление прошло успешно.

В результате всей работы мы создали полноценный класс DatabaseHelper, который позволяет управлять базой данных SQLite в Android-приложении. Этот класс расширяет SQLiteOpenHelper и инкапсулирует в себе все основные операции: создание базы, добавление, удаление, обновление и поиск контактов.

На рисунке 37 покажем первую часть итогового класса.

```

public class DatabaseHelper extends SQLiteOpenHelper {
    // Название таблицы и имена столбцов вынесены в отдельные переменные
    private static final String TABLE_NAME = "contacts";
    private static final String COLUMN_ID = "id";
    private static final String COLUMN_NAME = "name";
    private static final String COLUMN_PHONE = "phone";

    public DatabaseHelper(Context context) {
        super(context, "contacts.db", null, 1);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        String createTable = "CREATE TABLE " + TABLE_NAME + " (" +
            COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
            COLUMN_NAME + " TEXT, " +
            COLUMN_PHONE + " TEXT)";
        db.execSQL(createTable);
    }

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

```

Рисунок 37 – Класс DatabaseHelper ч.1

На рисунке 38 покажем вторую часть этого же класса.

```

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

// Добавление нового контакта
public boolean addContact(Contact contact) {
    SQLiteDatabase db = this.getWritableDatabase();
    ContentValues cv = new ContentValues();
    cv.put(COLUMN_NAME, contact.getName());
    cv.put(COLUMN_PHONE, contact.getPhone());
    long result = db.insert(TABLE_NAME, null, cv);
    db.close();
    return result != -1;
}

// Удаление контакта по номеру телефона
public boolean deleteContact(String phone) {
    SQLiteDatabase db = this.getWritableDatabase();
    int result = db.delete(TABLE_NAME, COLUMN_PHONE + " = ?", new String[]{phone});
    db.close();
    return result > 0;
}

// Поиск контакта по номеру телефона
public Contact findContact(String phone) {
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = db.query(TABLE_NAME,
        new String[]{COLUMN_ID, COLUMN_NAME, COLUMN_PHONE},
        COLUMN_PHONE + " = ?", new String[]{phone},
        null, null, null);

    if (cursor != null && cursor.moveToFirst()) {
        Contact contact = new Contact(cursor.getInt(0), cursor.getString(1), cursor.getString(2));
        cursor.close();
        db.close();
        return contact;
    }

    if (cursor != null) {
        cursor.close();
    }
    db.close();
    return null;
}

```

Рисунок 38 – Класс DatabaseHelper ч.2

На рисунке 39 покажем финальную третью часть кода этого же файла.

```

// Получение всех контактов
public List<Contact> getAllContacts() {
    List<Contact> contactList = new ArrayList<>();
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = db.rawQuery("SELECT * FROM " + TABLE_NAME, null);

    if (cursor.moveToFirst()) {
        do {
            Contact contact = new Contact(
                cursor.getInt(0),
                cursor.getString(1),
                cursor.getString(2));
            contactList.add(contact);
        } while (cursor.moveToNext());
    }

    cursor.close();
    db.close();
    return contactList;
}

// Обновление контакта
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);
    int result = db.update(TABLE_NAME, cv, COLUMN_PHONE + " = ?", new String[]{oldPhone});
    db.close();
    return result > 0;
}

```

Рисунок 39 – Класс DatabaseHelper ч.3

Затем мы создаём главную активность MainActivity, в которой отображается динамический список всех контактов. При добавлении, обновлении или удалении записи список будет автоматически изменяться на экране.

На рисунке 40 покажем первую часть MainActivity.

```

public class MainActivity extends AppCompatActivity {

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

        // Получаем ссылки на элементы интерфейса
        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);
        Button updateButton = findViewById(R.id.update_button);
        RecyclerView contactsList = findViewById(R.id.contacts_list);

        // Инициализируем базу данных и получаем список всех контактов
        DatabaseHelper dbHelper = new DatabaseHelper(this);
        List<Contact> contacts = dbHelper.getAllContacts();

        // Настраиваем адаптер и список
        ContactAdapter adapter = new ContactAdapter(contacts);
        contactsList.setLayoutManager(new LinearLayoutManager(this));
        contactsList.setAdapter(adapter);

        // Добавление нового контакта
        saveButton.setOnClickListener(v -> {
            String name = nameInput.getText().toString();
            String phone = phoneInput.getText().toString();

            if (dbHelper.addContact(new Contact(0, name, phone))) {
                contacts.add(new Contact(0, name, phone));
                adapter.notifyItemInserted(contacts.size() - 1);
                Toast.makeText(this, "Контакт успешно сохранён!", Toast.LENGTH_SHORT).show();
            } else {
                Toast.makeText(this, "Не удалось сохранить контакт", Toast.LENGTH_SHORT).show();
            }
        });
    }
}

```

Рисунок 40 – Класс MainActivity ч.1

На рисунке 41 покажем вторую часть кода MainActivity.

```

// Удаление контакта
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(this, "Контакт удалён!", Toast.LENGTH_SHORT).show();
        } else {
            Toast.makeText(this, "Контакт не найден", Toast.LENGTH_SHORT).show();
        }
    } else {
        Toast.makeText(this, "Ошибка при удалении", Toast.LENGTH_SHORT).show();
    }
});

// Поиск контакта 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(this, "Контакт найден: " + foundContact.getName(), Toast.LENGTH_SHORT).show();
    } else {
        Toast.makeText(this, "Контакт не найден", Toast.LENGTH_SHORT).show();
    }
});

```

Рисунок 41 – Класс MainActivity ч.2

На рисунке 42 покажем третью и заключительную часть класса MainActivity.

```

// Обновление контакта
updateButton.setOnClickListener(v -> {
    String oldPhone = phoneInput.getText().toString(); // старый номер
    String newName = nameInput.getText().toString(); // новое имя
    String newPhone = phoneInput.getText().toString(); // новый номер

    if (dbHelper.updateContact(oldPhone, newName, newPhone)) {
        Toast.makeText(this, "Контакт обновлён!", Toast.LENGTH_SHORT).show();
        refreshContactsList(dbHelper, contacts, adapter, contactsList);
    } else {
        Toast.makeText(this, "Не удалось обновить контакт", Toast.LENGTH_SHORT).show();
    }
});
}

// Метод для обновления списка после изменений в базе
private void refreshContactsList(DatabaseHelper dbHelper, List<Contact> contacts, ContactAdapter adapter, RecyclerView contactsList) {
    contacts = dbHelper.getAllContacts(); // Загружаем актуальный список
    adapter = new ContactAdapter(contacts); // Создаём новый адаптер
    contactsList.setAdapter(adapter); // Устанавливаем его
}
}

```

Рисунок 42 – Класс MainActivity ч.3

Так как мы используем RecyclerView для отображения списка контактов, нам необходимо реализовать собственный адаптер, который будет отвечать за отображение каждого элемента и обновление данных на экране при изменениях в базе.

На рисунке 43 покажем код класса ContactAdapter.



```

public class ContactAdapter extends RecyclerView.Adapter<ContactAdapter.ViewHolder> {
    private final List<Contact> contacts;

    // Класс ViewHolder описывает, как отображаются отдельные элементы списка
    public static class ViewHolder extends RecyclerView.ViewHolder {
        private final TextView nameTextView;
        private final TextView phoneTextView;

        public ViewHolder(View view) {
            super(view);
            nameTextView = view.findViewById(R.id.contact_name);
            phoneTextView = view.findViewById(R.id.contact_phone);
        }

        // Привязываем данные контакта к элементам интерфейса
        public void bind(Contact contact) {
            nameTextView.setText(contact.getName());
            phoneTextView.setText(contact.getPhone());
        }
    }

    // Конструктор адаптера принимает список контактов
    public ContactAdapter(List<Contact> contacts) {
        this.contacts = contacts;
    }

    @NonNull
    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        // Загружаем макет отдельного элемента списка
        View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.item_contact, parent, false);
        return new ViewHolder(view);
    }

    @Override
    public void onBindViewHolder(ViewHolder holder, int position) {
        // Передаём данные текущего контакта в ViewHolder
        holder.bind(contacts.get(position));
    }

    @Override
    public int getItemCount() {
        // Возвращаем общее количество элементов в списке
        return contacts.size();
    }
}

```

Рисунок 43 – Класс ContactAdapter

На рисунке 44 покажем работоспособность нашей программы.



Рисунок 44 – Работоспособность программы

Даже если перезапустить приложение, то при открытии будет выведен список сохраненных ранее контактов.

Таким образом, получилось готовое приложение для хранения телефонных контактов.

## 2 ПРАКТИЧЕСКАЯ ЧАСТЬ

### 2.1 Реализация записи, получения, изменения и удаления имени пользователя приложения через SharedPreferences

Для решения данной задачи на рисунке 1 покажем разметку и интерфейс для activity\_main.xml.

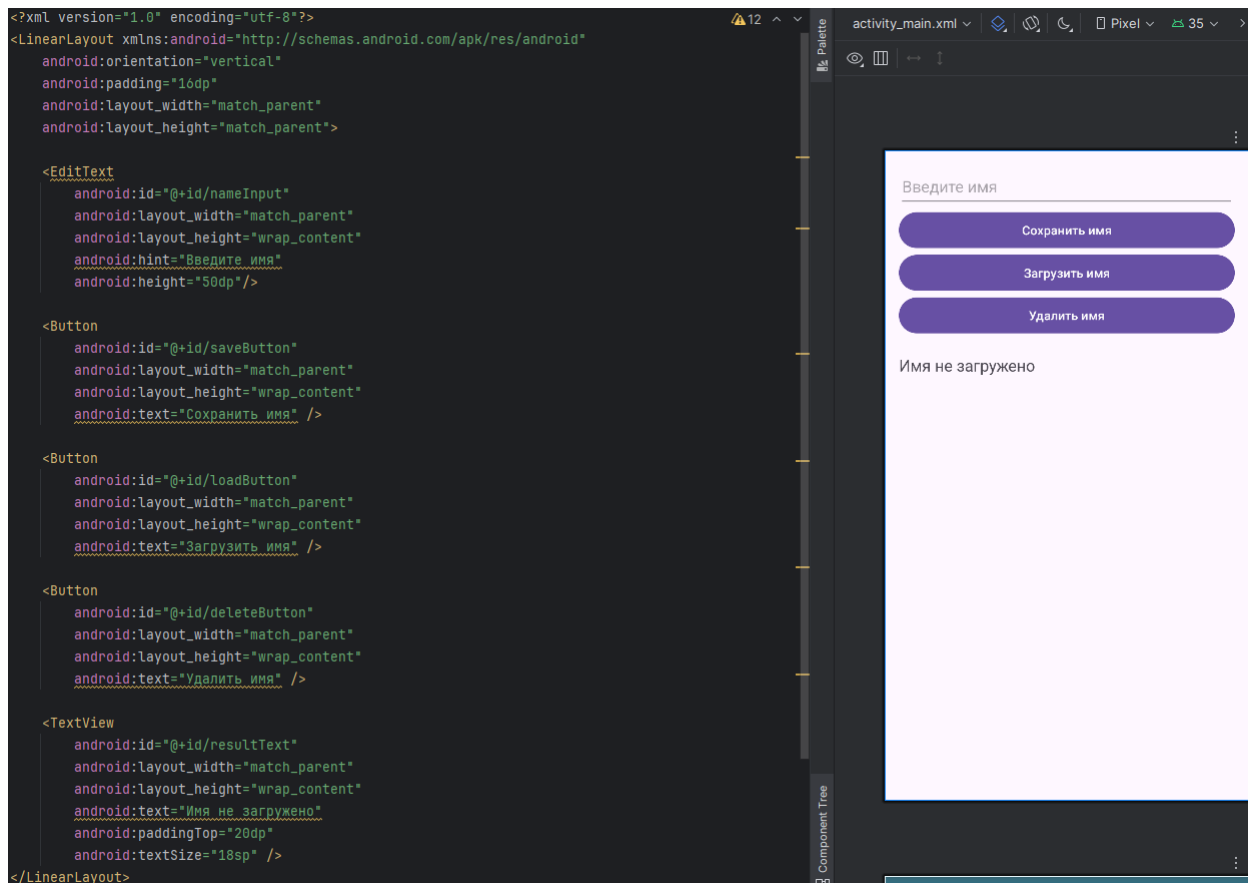


Рисунок 45 – Разметка и интерфейс activity\_main.xml

На рисунке 46 покажем код MainActivity.

```

public class MainActivity extends AppCompatActivity {
    2 usages
    EditText nameInput;
    4 usages
    TextView resultText;
    2 usages
    Button saveButton, loadButton, deleteButton;
    1 usage
    private static final String PREFS_NAME = "UserPrefs";
    3 usages
    private static final String KEY_NAME = "userName";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        nameInput = findViewById(R.id.nameInput);
        resultText = findViewById(R.id.resultText);
        saveButton = findViewById(R.id.saveButton);
        loadButton = findViewById(R.id.loadButton);
        deleteButton = findViewById(R.id.deleteButton);
        SharedPreferences prefs = getSharedPreferences(PREFS_NAME, MODE_PRIVATE);
        saveButton.setOnClickListener( View v -> {
            String name = nameInput.getText().toString();
            prefs.edit().putString(KEY_NAME, name).apply();
            Toast.makeText( context: this, text: "Имя сохранено", Toast.LENGTH_SHORT).show();
        });

        loadButton.setOnClickListener( View v -> {
            String name = prefs.getString(KEY_NAME, defValue: null);
            if (name != null) {
                resultText.setText("Сохранённое имя: " + name);
            } else {
                resultText.setText("Имя не найдено");
            }
        });

        deleteButton.setOnClickListener( View v -> {
            prefs.edit().remove(KEY_NAME).apply();
            resultText.setText("Имя удалено");
        });
    }
}

```

Рисунок 46 – Код MainActivity

На рисунке 47 покажем работоспособность программы.

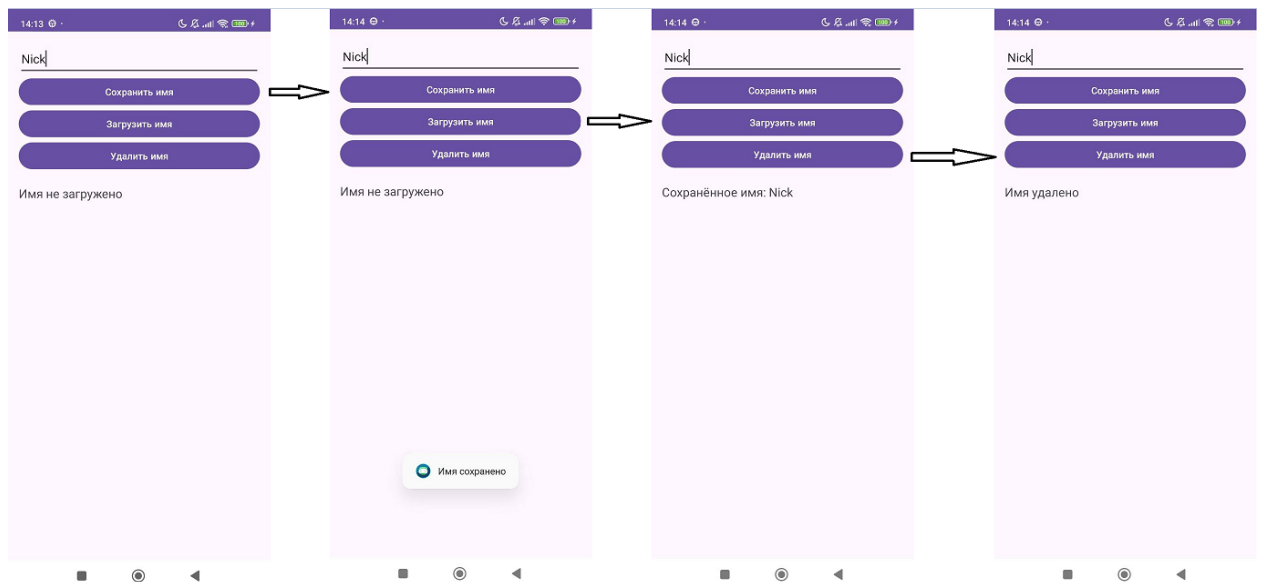


Рисунок 47 – Работоспособность программы

## 2.2 Создание тематической базы данных. Реализация методов записи, поиска, изменения и удаления данных

Для решения данной задачи напомним класс Car, который будет иметь 5 полей, как и требует того задание. Первую часть кода класса Car покажем на рисунке 48.

```

public class Car {
    3 usages
    private int id;
    3 usages
    private String modelName;
    3 usages
    private String manufacturer;
    3 usages
    private int year;
    3 usages
    private double price;

    1 usage
    public Car(int id, String modelName, String manufacturer, int year, double price) {
        this.id = id;
        this.modelName = modelName;
        this.manufacturer = manufacturer;
        this.year = year;
        this.price = price;
    }

    no usages
    public Car(String modelName, String manufacturer, int year, double price) {
        this(id: -1, modelName, manufacturer, year, price);
    }

    no usages
    public double getPrice() {
        return price;
    }

    no usages
    public void setPrice(double price) {
        this.price = price;
    }

    no usages
    public String getManufacturer() {
        return manufacturer;
    }
}

```

Рисунок 48 – Класс Car ч.1

На рисунке 49 покажем вторую часть кода класса Car.

```

no usages
public void setManufacturer(String manufacturer) {
    this.manufacturer = manufacturer;
}

no usages
public String getModelName() {
    return modelName;
}

no usages
public void setModelName(String modelName) {
    this.modelName = modelName;
}

no usages
public int getId() {
    return id;
}

no usages
public void setId(int id) {
    this.id = id;
}

no usages
public int getYear() {
    return year;
}

no usages
public void setYear(int year) {
    this.year = year;
}
}

```

Рисунок 49 – Класс Car ч.2

Далее напишем класс CarDatabaseHelper. На рисунке 50 покажем первую часть кода этого класса.

```

public class CarDatabaseHelper extends SQLiteOpenHelper {
    1 usage
    private static final String DB_NAME = "car_dealer.db";
    1 usage
    private static final int DB_VERSION = 1;

    no usages
    public CarDatabaseHelper(Context context) {
        super(context, DB_NAME, factory: null, DB_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        String query = "CREATE TABLE cars (" +
            "id INTEGER PRIMARY KEY AUTOINCREMENT," +
            "modelName TEXT," +
            "manufacturer TEXT," +
            "year INTEGER," +
            "price REAL)";
        db.execSQL(query);
    }

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

    no usages
    public void insertCar(Car car) {
        SQLiteDatabase db = getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put("modelName", car.getModelName());
        values.put("manufacturer", car.getManufacturer());
        values.put("year", car.getYear());
        values.put("price", car.getPrice());
        db.insert(table: "cars", nullColumnHack: null, values);
    }
}

```

Рисунок 50 – Класс CarDatabaseHelper ч.1

На рисунке 51 покажем вторую часть кода этого же класса.



```

no usages
public Cursor getAllCars() {
    SQLiteDatabase db = getReadableDatabase();
    return db.rawQuery( sql: "SELECT * FROM cars", selectionArgs: null);
}

no usages
public void updateCar(Car car) {
    SQLiteDatabase db = getWritableDatabase();
    ContentValues values = new ContentValues();
    values.put("modelName", car.getModelName());
    values.put("manufacturer", car.getManufacturer());
    values.put("year", car.getYear());
    values.put("price", car.getPrice());
    db.update( table: "cars", values, whereClause: "id = ?", new String[]{String.valueOf(car.getId())});
}

no usages
public void deleteCar(int id) {
    SQLiteDatabase db = getWritableDatabase();
    db.delete( table: "cars", whereClause: "id = ?", new String[]{String.valueOf(id)});
}
}

```

Рисунок 51 – Класс CarDatabaseHelper ч.2

Теперь перепишем разметку для activity\_main.xml под новые условия и покажем результат на рисунке 52.

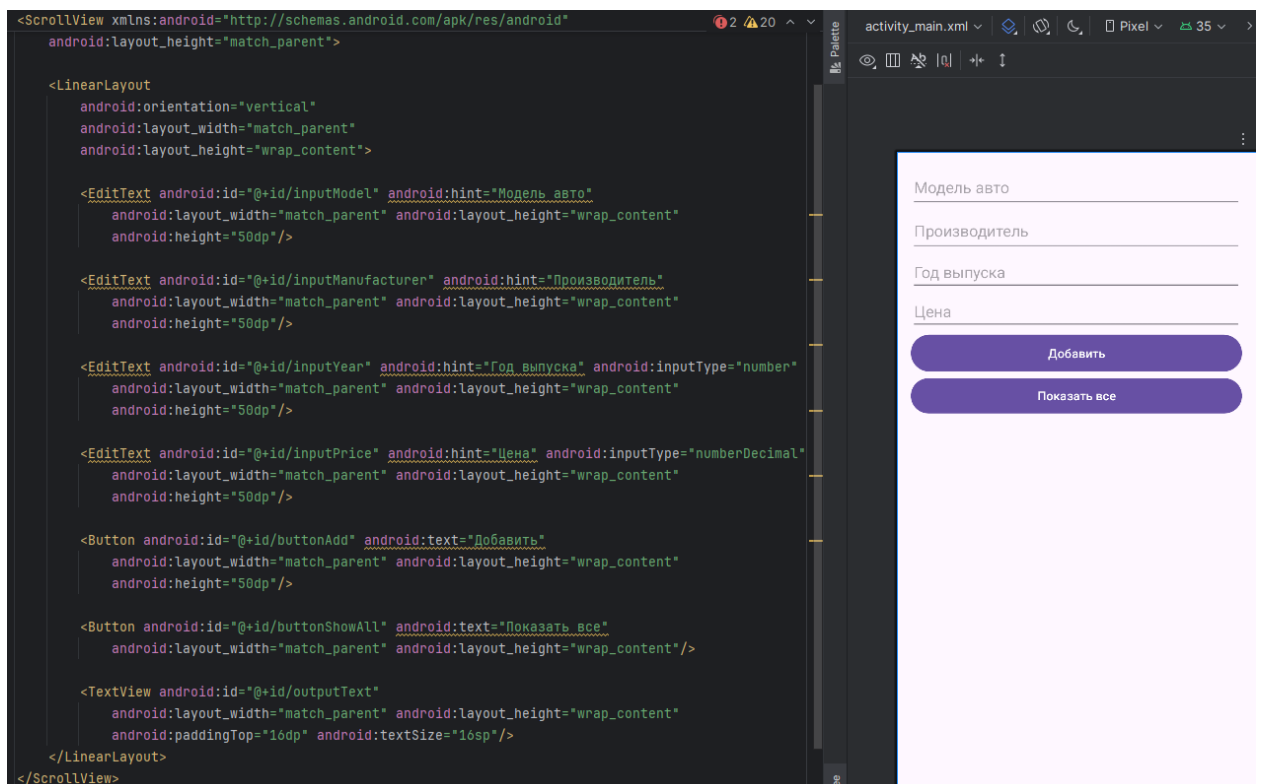


Рисунок 52 – Разметка и интерфейс activity\_main.xml

Далее на рисунке 53 покажем первую часть кода класса MainActivity.

```
public class MainActivity extends AppCompatActivity {  
    EditText inputModel, inputManufacturer, inputYear, inputPrice;  
    2 usages  
    TextView outputText;  
    2 usages  
    Button buttonAdd, buttonShowAll;|  
    3 usages  
    CarDatabaseHelper dbHelper;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        dbHelper = new CarDatabaseHelper(context, this);  
  
        inputModel = findViewById(R.id.inputModel);  
        inputManufacturer = findViewById(R.id.inputManufacturer);  
        inputYear = findViewById(R.id.inputYear);  
        inputPrice = findViewById(R.id.inputPrice);  
        outputText = findViewById(R.id.outputText);  
        buttonAdd = findViewById(R.id.buttonAdd);  
        buttonShowAll = findViewById(R.id.buttonShowAll);  
  
        buttonAdd.setOnClickListener( View v -> {  
            String model = inputModel.getText().toString();  
            String manufacturer = inputManufacturer.getText().toString();  
            int year = Integer.parseInt(inputYear.getText().toString());  
            double price = Double.parseDouble(inputPrice.getText().toString());  
  
            dbHelper.insertCar(new Car(model, manufacturer, year, price));  
            Toast.makeText(context, this, text: "Добавлено", Toast.LENGTH_SHORT).show();  
        });  
    }  
}
```

Рисунок 53 – Класс MainActivity ч.1

На рисунке 54 покажем вторую часть кода класса MainActivity.

```

buttonShowAll.setOnClickListener( View v -> {
    Cursor cursor = dbHelper.getAllCars();
    StringBuilder builder = new StringBuilder();
    while (cursor.moveToNext()) {
        builder.append("ID: ").append(cursor.getInt( columnIndex: 0)).append("\n")
            .append("Модель: ").append(cursor.getString( columnIndex: 1)).append("\n")
            .append("Производитель: ").append(cursor.getString( columnIndex: 2)).append("\n")
            .append("Год: ").append(cursor.getInt( columnIndex: 3)).append("\n")
            .append("Цена: ").append(cursor.getDouble( columnIndex: 4)).append("\n\n");
    }
    outputText.setText(builder.toString());
});
}
}

```

Рисунок 54 – Класс MainActivity ч.2

На рисунке 55 покажем работоспособность программы.

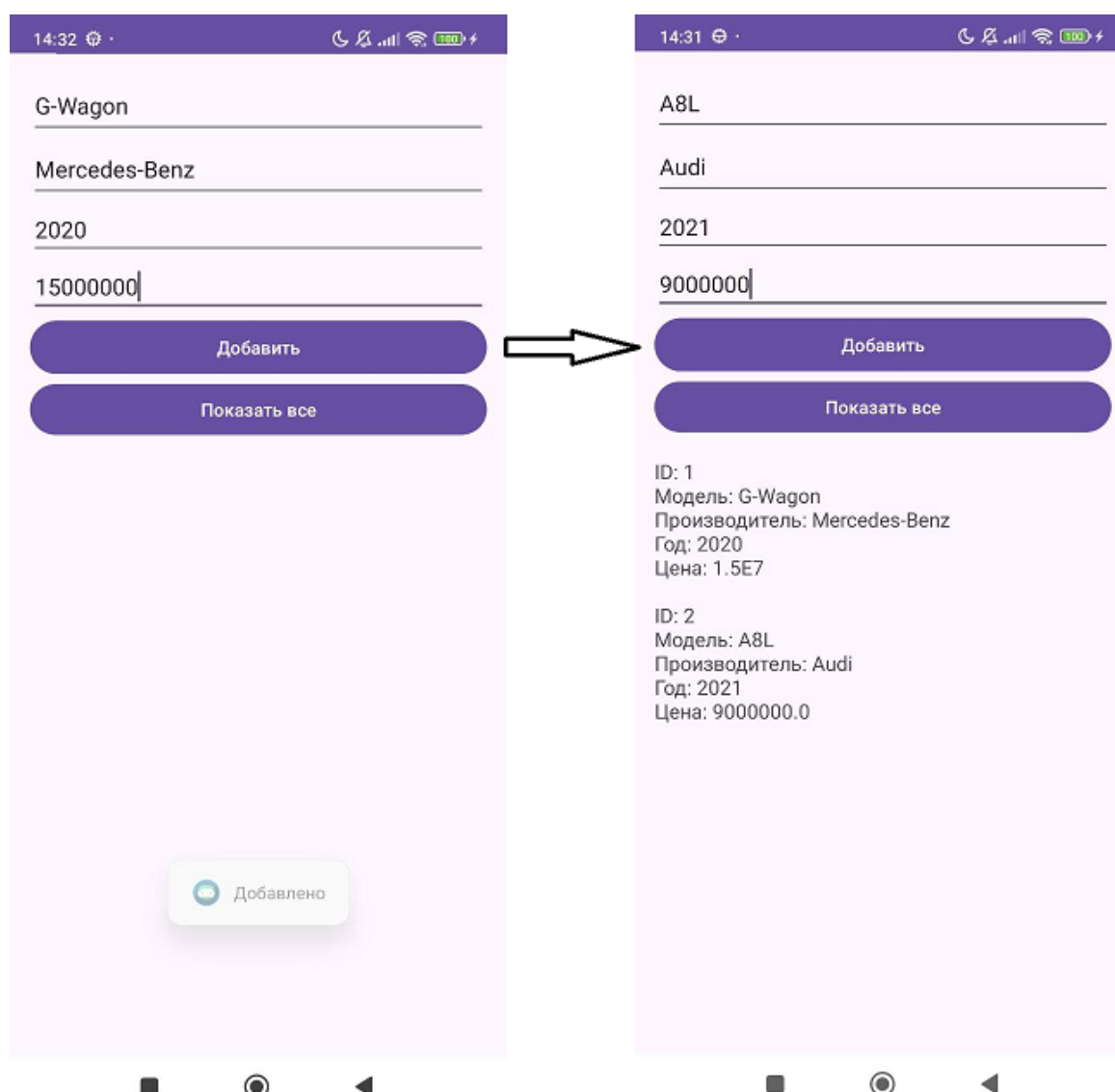


Рисунок 55 – Работоспособность программы

## **ЗАКЛЮЧЕНИЕ**

В ходе выполнения практической работы были успешно реализованы два способа локального хранения данных в Android: SharedPreferences и база данных SQLite.

Была создана форма для взаимодействия с пользователем, позволяющая сохранять и редактировать имя пользователя с помощью SharedPreferences, что продемонстрировало удобство работы с настройками приложения.

Кроме того, была спроектирована структура базы данных на тему автосалона, где каждая запись содержит информацию о модели автомобиля, производителе, годе выпуска и цене. Пользователь получил возможность выполнять основные операции: добавление, просмотр, изменение и удаление записей.

Практика показала, как комбинировать разные подходы к хранению информации в Android, в зависимости от объёма и характера данных, что является важным шагом к созданию полноценных и устойчивых приложений.