

Часть 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.

Зачем он нужен?

- **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 объемов данных без необходимости создавать и управлять базой данных.

Для использования SharedPreferences, вам 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.

```
// Сохранение данных
```

```

SharedPreferences                                sharedPreferences
getSharedPreferences("myPreferences", MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPreferences.edit();

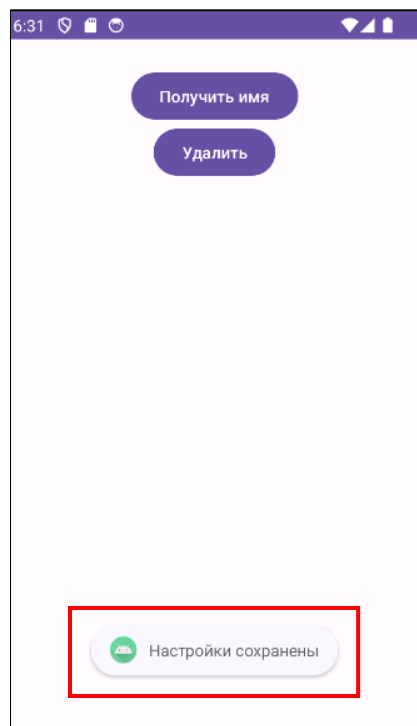
// Сохранение строкового значения
editor.putString("username", "User123");
editor.putInt("sessionCount", 5);
editor.putBoolean("loggedIn", true);

// Сохранение изменений
editor.apply();

```

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.



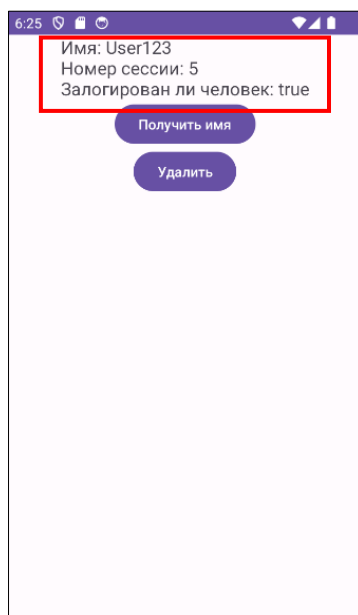
```

// Получение данных
SharedPreferences                                sharedPreferences
getSharedPreferences("myPreferences", MODE_PRIVATE);
// Чтение строкового значения

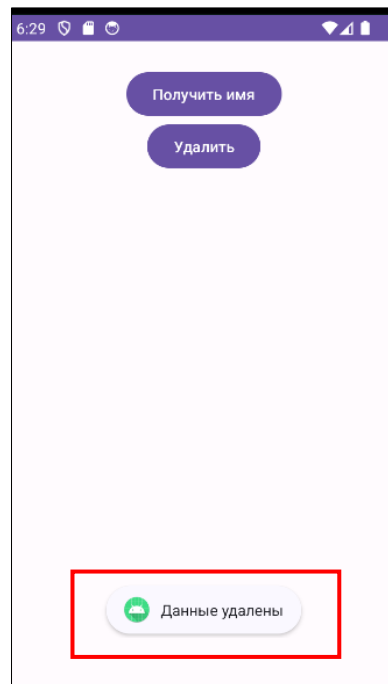
```

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

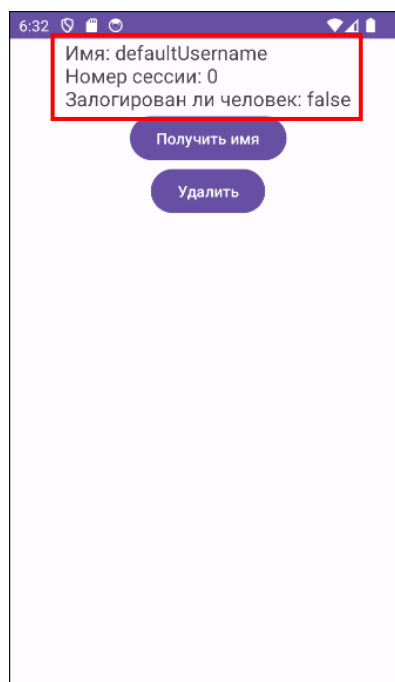
При чтении строковых значений, метод **getString (String key, String defValue)** возвращает из настроек значение типа String, которое имеет ключ key. Если элемента с таким ключом не окажется, то возвращается значение defValue, передаваемое вторым параметром.



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



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



Часть 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:

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

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

1. **Таблицы:** Данные хранятся в таблицах, где каждая таблица обычно представляет собой одну сущность или объект (например, клиенты, заказы, продукты).
2. **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.
3. **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.
4. **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.
5. **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.

6. 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).

Часть 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 следующий синтаксис:

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

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.

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

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:

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

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 следующий синтаксис:

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

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.

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

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:

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

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.

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

Для получения данных в SQLite применяется команда **SELECT**. В упрощенном виде она имеет следующий синтаксис:

```
SELECT список_столбцов FROM имя_таблицы
```

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:

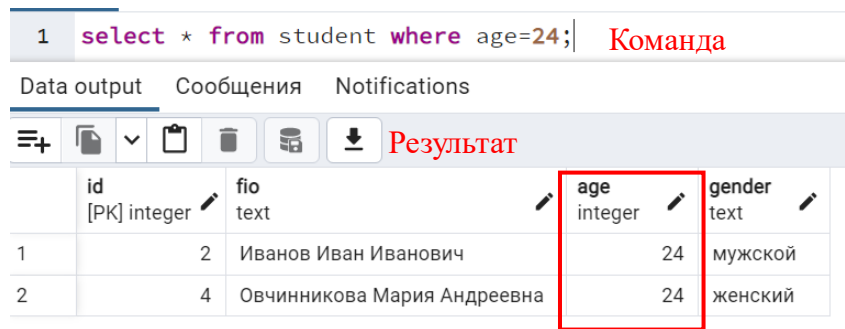
```
SELECT список_столбцов FROM имя_таблицы WHERE условие
```

- **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:

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

Символ * означает, что я хочу выбрать все колонки (можно выбирать конкретные, а можно сразу все).



The screenshot shows a database management interface. At the top, a SQL query is entered in a text box: `1 select * from student where age=24;`. To the right of the query is a red label "Команда". Below the query box are three tabs: "Data output", "Сообщения", and "Notifications". Below the tabs is a toolbar with icons for various actions. To the right of the toolbar is a red label "Результат". Below the toolbar is a table with the following data:

	id [PK] integer	fio text	age integer	gender text
1	2	Иванов Иван Иванович	24	мужской
2	4	Овчинникова Мария Андреевна	24	женский

Если бы у меня была не база данных, а простые excel-файлики, то же действие было бы:

1. Открыть файл с нужными данными (student)
2. 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 следующий синтаксис:

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

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

<pre>UPDATE student SET age=21 WHERE gender='женский'; select * from student;</pre>			
output Сообщения Notifications			
id [PK] integer	fio text	age integer	gender 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

Команда **DELETE** удаляет данные из БД. Она имеет следующий формальный синтаксис:

```
DELETE FROM название_таблицы WHERE условие_удаления
```

Удалим из нашей таблицы со студентами тех, кому больше 23 лет:

<pre>DELETE FROM student WHERE age>23; select * from student;</pre>			
output Сообщения Notifications			
id [PK] integer	fio text	age integer	gender text
3	Новичков Дмитрий Евгеньевич	22	мужской
1	Муравьёва Екатерина Андреев...	21	женский
4	Овчинникова Мария Андреевна	21	женский
Запись удалилась из базы данных			

Если необходимо вовсе удалить все строки вне зависимости от условия, то условие можно не указывать:

```
DELETE FROM название_таблицы
```

id [PK] integer	fio text	age integer	gender text
Все данные удалены из БД			

Для удаления таблицы полностью используется команда **DROP TABLE**, которая имеет синтаксис:

```
DROP TABLE название_таблицы
```

```
1 DROP TABLE student;
2 select * from student;
```

Data output Сообщения Notifications

ERROR: ОШИБКА: отношение "student" не существует
LINE 1: select * from student;
 ^

При попытке считать данные, выдает ошибку, так как БД была удалена

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:

```
1 DROP TABLE IF EXISTS student;
```

Data output Сообщения Notifications

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

Часть 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 SQL-запросы.

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.

```
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;
    }

    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;
    }
}

```

Далее необходимо создать отдельный класс помощника DatabaseHelper 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.

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

```

@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);
}

```

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.

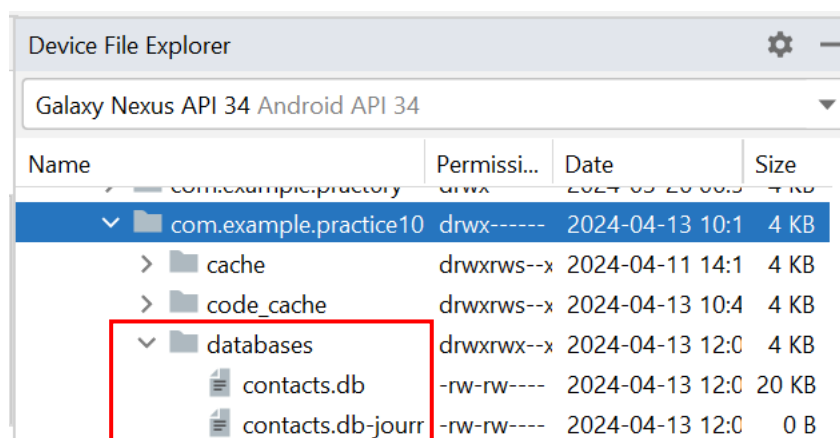
```

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

```

В данном случае для примера использован примитивный поход 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 т.д.

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**.



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:

```
// 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;
}
```

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()**.

```
// 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;
}
```

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.

```
// 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;
}

```

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

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 работу курсор должен быть закрыт методом **close()**.

Можно получить сразу все данные из базы данных, отобразив их в список.

```

// 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;
}

```


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 бд. Если этот метод возвратит значение **false**, значит запрос не получил никаких данных из бд.

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

```
//Обновляем данные
    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, COLUMN_PHONE + " =
?", new String[]{oldPhone});
        db.close();
        return result > 0;
    }
}
```

Итоговый класс выглядит следующим образом:

```
public class DatabaseHelper extends SQLiteOpenHelper {
    //Пропишем в отдельные переменные название БД и её столбов
    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);
    }

    // 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;
        }

        // 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;
        }

```

```

        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, COLUMN_PHONE + " =
?", new String[]{oldPhone});
            db.close();
            return result > 0;
        }
    }
}

```

Затем создаем главную активность, на экране которой будет динамический список со всеми контактами. При добавлении, изменении или удалении данных, список на экране будет изменяться.

```

public class MainActivity extends AppCompatActivity {

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

        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(this);
        List<Contact> contacts = dbHelper.getAllContacts();
    }
}

```

```

        ContactAdapter adapter = new ContactAdapter(contacts);
        contactsList.setLayoutManager(new
LinearLayoutManager(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(0, name, phone)))
{
                contacts.add(new Contact(0, name, phone));
                adapter.notifyItemInserted(contacts.size() - 1);
                Toast.makeText(this, "Contact saved
successfully!", Toast.LENGTH_SHORT).show();
            } else {
                Toast.makeText(this, "Failed to save contact",
Toast.LENGTH_SHORT).show();
            }
        });
        //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(this, "Contact deleted
successfully!", Toast.LENGTH_SHORT).show();

```

```

        } else {
            Toast.makeText(this, "Contact not found",
Toast.LENGTH_SHORT).show();
        }
    } else {
        Toast.makeText(this, "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(this, "Contact found: " +
foundContact.getName(), Toast.LENGTH_SHORT).show();
    } else {
        Toast.makeText(this, "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(this, "Contact updated
successfully!", Toast.LENGTH_SHORT).show();
    }
});

```

```

        // Обновляем список и адаптер
        refreshContactsList(dbHelper, contacts, adapter,
contactsList);

        } else {
            Toast.makeText(this, "Failed to update contact",
Toast.LENGTH_SHORT).show();
        }
    });

}

// Метод для обновления списка контактов после изменения в
базе данных
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);
}
}

```

Так как мы используем RecyclerView, то нам также необходимо прописать код адаптера для обновления данных в списке.

```

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

    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);  
    }  
  
    public void bind(Contact contact) {  
        nameTextView.setText(contact.getName());  
        phoneTextView.setText(contact.getPhone());  
    }  
}  
  
public ContactAdapter(List<Contact> contacts) {  
    this.contacts = contacts;  
}  
  
@NonNull  
@Override  
public ViewHolder onCreateViewHolder(ViewGroup parent, int  
viewType) {  
    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)  
{  
    holder.bind(contacts.get(position));  
}  
  
@Override  
public int getItemCount() {  
    return contacts.size();  
}  
}
```


3:45

Kate

12345678

Save

Update Contact

Delete

Find

3:46

Kate

12345678

Save

Update Contact

Delete

Find

Kate 12345678

3:49

Name

12345678

Save

Update Contact

Delete

Find

Natali 987654321

Ivan 98765432101

Contact deleted successfully!

3:48

Name

12345678

Save

Update Contact

Delete

Find

Kate 12345678

Natali 987654321

Ivan 98765432101

3:47

Kate

12345678

Save

Update Contact

Delete

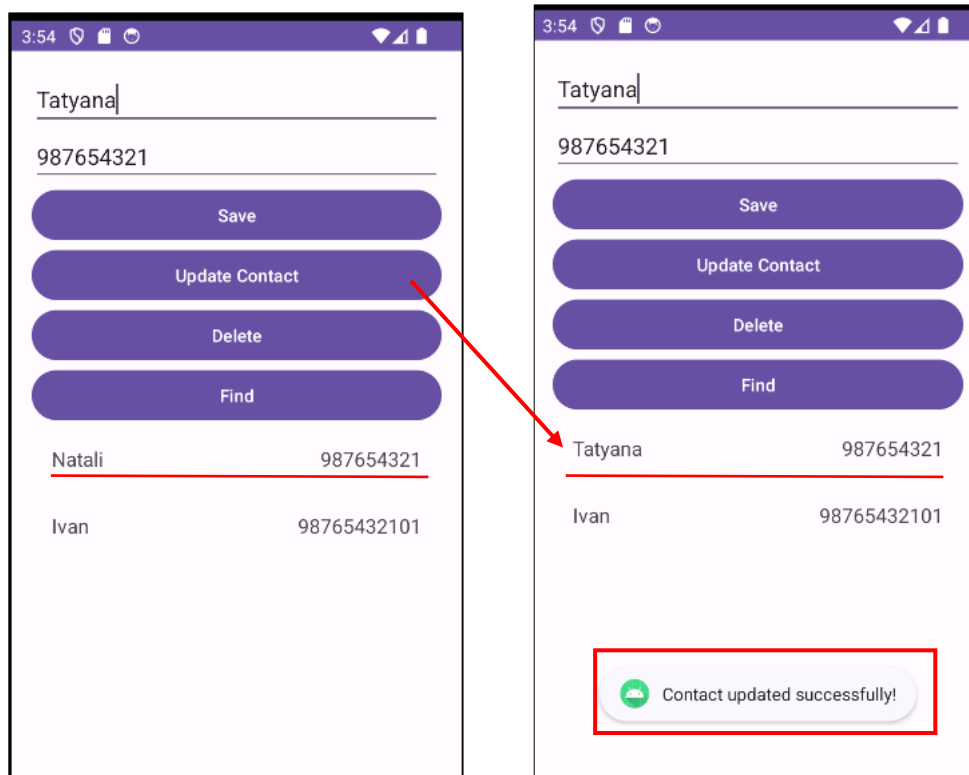
Find

Kate 12345678

Natali 987654321

Ivan 98765432101

Contact found: Kate



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

Таким образом, получилось готовое приложение для хранения телефонных контактов.

Задание

1. Реализовать запись, получение, изменение и удаление имени пользователя приложения через SharedPreferences.
2. Создать базу данных по выбранной тематике. Реализовать методы записи, поиска, изменения и удаления данных. В базе данных должна быть минимум одна таблица с 5 полями.

Источники

- 1) <https://developer.android.com/training/data-storage/sqlite>
- 2) <https://metanit.com/java/android/14.5.php?ysclid=luv1bkqgxe543443786>
- 3) <https://startandroid.ru/ru/uroki/vse-uroki-spiskom/74-urok-34-hranenie-dannyh-sqlite.html>
- 4) <https://www.geeksforgeeks.org/how-to-create-and-add-data-to-sqlite-database-in-android/>
- 5) <https://developer.android.com/training/data-storage/shared-preferences>
- 6) <https://metanit.com/java/android/12.1.php?ysclid=luv1f9movr257519492>