|  |  |  |
| --- | --- | --- |
|  |  |  |
| 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 группыИКБО-20-23 | Комисарик М.А. |
| **Проверил:**  Старший преподаватель кафедры МОСИТ | Шешуков Л.С. |

Москва 2025 г.

СОДЕРЖАНИЕ

**РТУ МИРЭА** 1

СОДЕРЖАНИЕ 2

[1 ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ 3](#_Toc197129707)

[1.1 Сохранение состояния приложения 3](#_Toc197129708)

[1.2 Хранение данных в Android 4](#_Toc197129709)

[1.3 Внутреннее хранилище 6](#_Toc197129710)

[1.3.1 Создание и запись файла 6](#_Toc197129711)

[1.3.2 Чтение файла 7](#_Toc197129712)

[1.3.3 Удаление файла 8](#_Toc197129713)

[1.4 Внешнее хранилище 9](#_Toc197129714)

[1.4.1 Создание и запись файла 10](#_Toc197129715)

[1.4.2 Чтение файла 11](#_Toc197129716)

[1.4.3 Удаление файла 11](#_Toc197129717)

[2 ПРАКТИЧЕСКОЕ ЗАДАНИЕ 12](#_Toc197129718)

[2.1 Внутреннее хранилище 12](#_Toc197129719)

[2.1.1 Разметка 12](#_Toc197129720)

[2.1.2 Реализация работы с файлами 14](#_Toc197129721)

[2.1.3 Сохранение состояния приложения 17](#_Toc197129722)

[2.1.4 Тестирование 17](#_Toc197129723)

[2.2 Работа с внешней памятью 20](#_Toc197129724)

[2.2.1 Разрешения 20](#_Toc197129725)

[2.2.2 Разметка 20](#_Toc197129726)

[2.2.3 Реализация 21](#_Toc197129727)

[2.2.4 Тестирование 23](#_Toc197129728)

[ЗАКЛЮЧЕНИЕ 25](#_Toc197129729)

# ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ

## Класс SharedPreference

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 введённые 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).

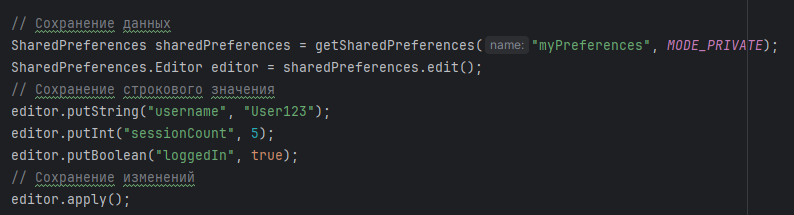


Рисунок – Работа с экземпляром класса SharedPreferences

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).

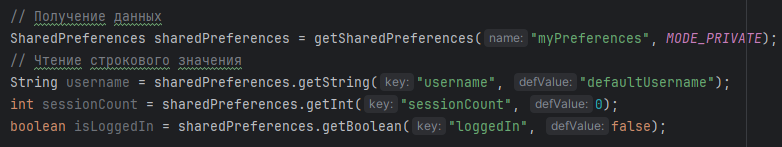


Рисунок – Получение данных из экземпляра класса SharedPreferences

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

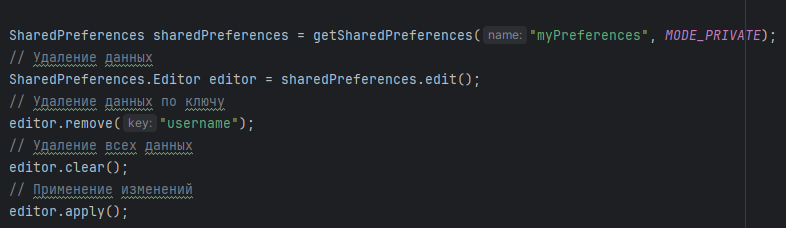


Рисунок – Удаление данных из настроек

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

## 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).



Рисунок – 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).

## 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).

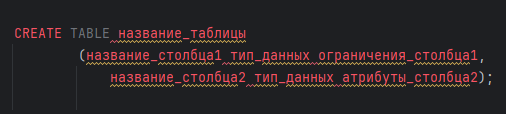


Рисунок – Синтаксис создания таблицы в SQL

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).

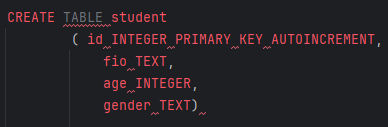


Рисунок – Создание таблицы student

**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).

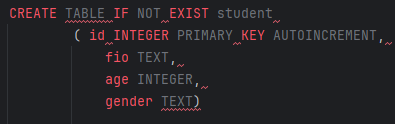


Рисунок – Реализация устранения ошибки при повторном создании существующей таблицы

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).



Рисунок – 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).



Рисунок – Добавление элемента в таблицу

После названия таблицы указаны столбцы, в которые мы хотим выполнить добавление данные – (фио, 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).

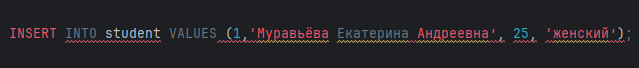


Рисунок – Добавление элемента без строгого указания стобцов

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).

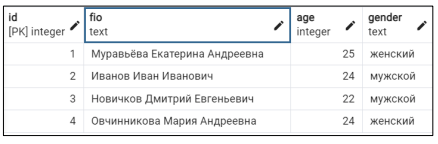


Рисунок – Расширенная версия таблицы

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



Рисунок – 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).



Рисунок – Указание условия при использовании команды SELECT

Описать контекст рисунка Рисунок 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).



Рисунок – Конкретизация условия WHERE

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

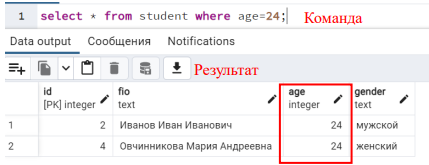


Рисунок – Результат конкретизированного поиска

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).



Рисунок – Синтаксис команды UPDATE

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

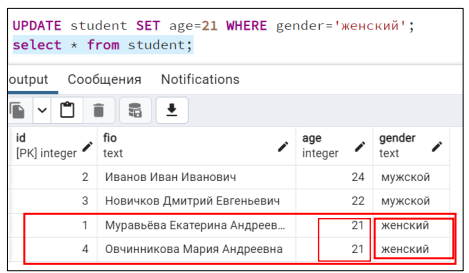


Рисунок – Установка конкретного возраста в таблице

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



Рисунок – Синтаксис команды DELETE

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

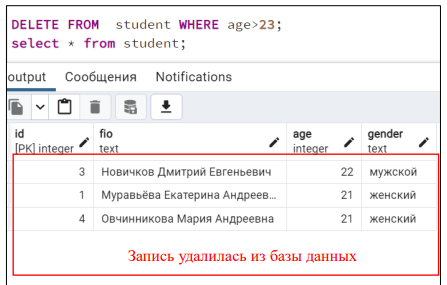


Рисунок – Результат удаления записей

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



Рисунок – 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).

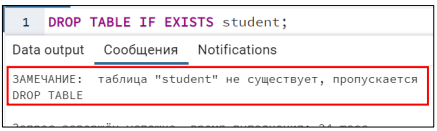


Рисунок – Способ избегания ошибки при удалении несуществующей таблицы

## 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 хранится данные о контактах (Рисунок 22).

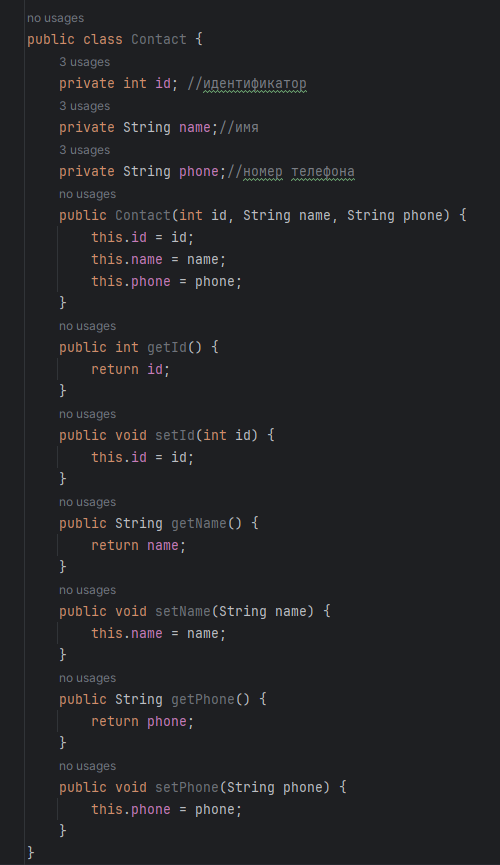


Рисунок – Описание класса Contract

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.



Рисунок – 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 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 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).

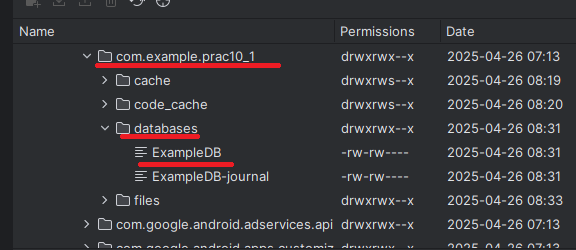


Рисунок – Расположение созданной базы данных

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 будет записывать данные в бд, то воспользуемся вторым методом (Рисунок 25).

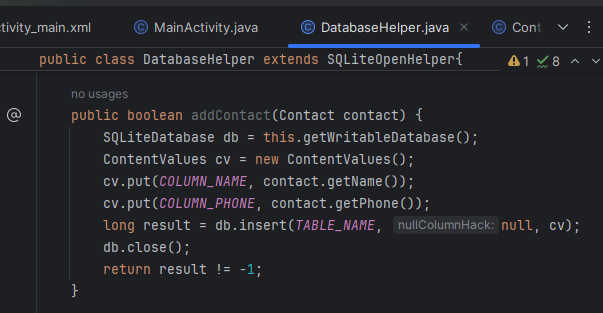


Рисунок – Описание метода addContact()

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).

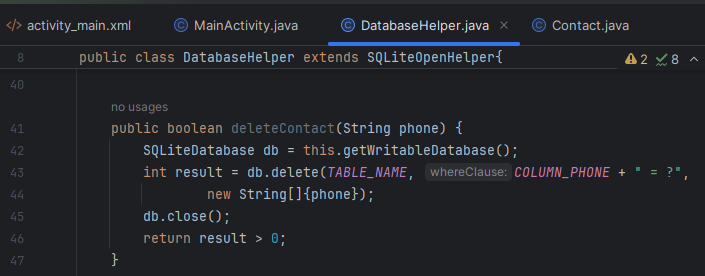


Рисунок – Описание метода deleteContact()

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).

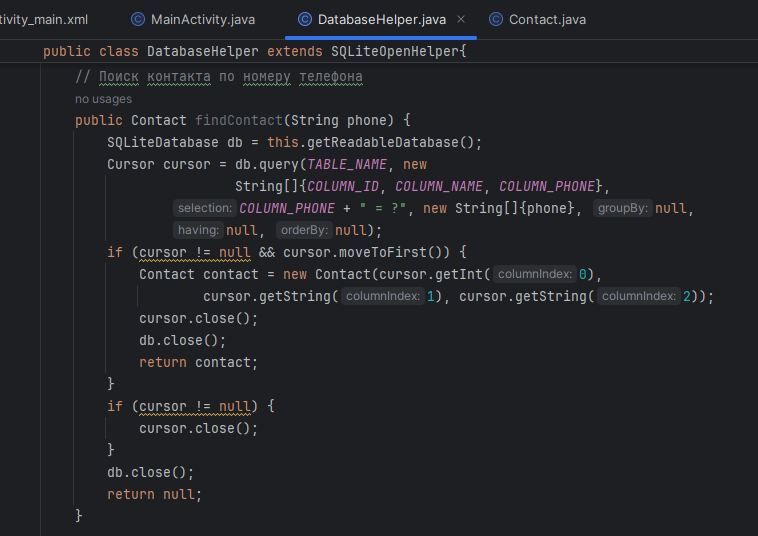


Рисунок – Описание метода поиска контакта в базе данных

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 работу курсор должен быть закрыт методом close().

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

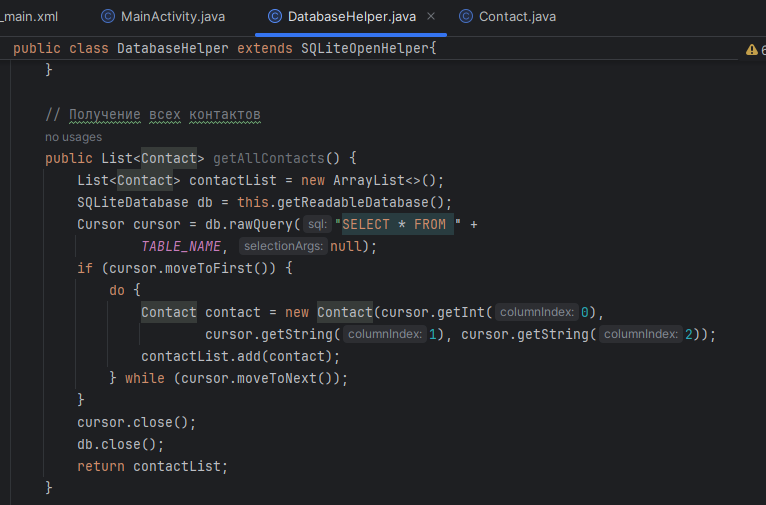


Рисунок – Описание метода поиска всех возможных контактов в базе данных

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).

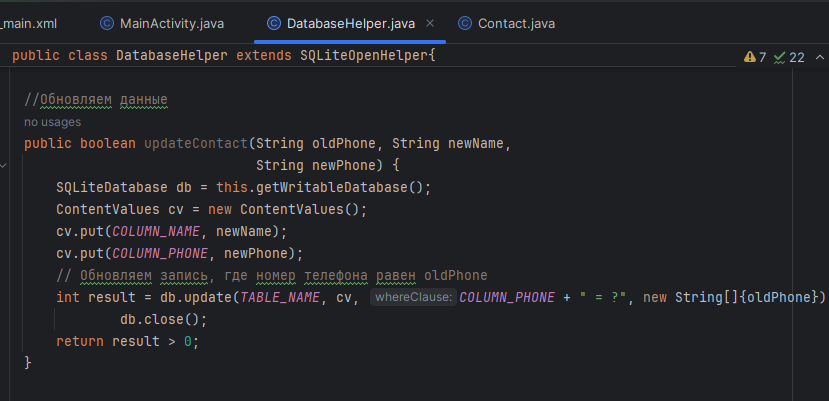


Рисунок – Описание метода обновления контакта в базе данных

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 Рисунок 30-Рисунок 32).

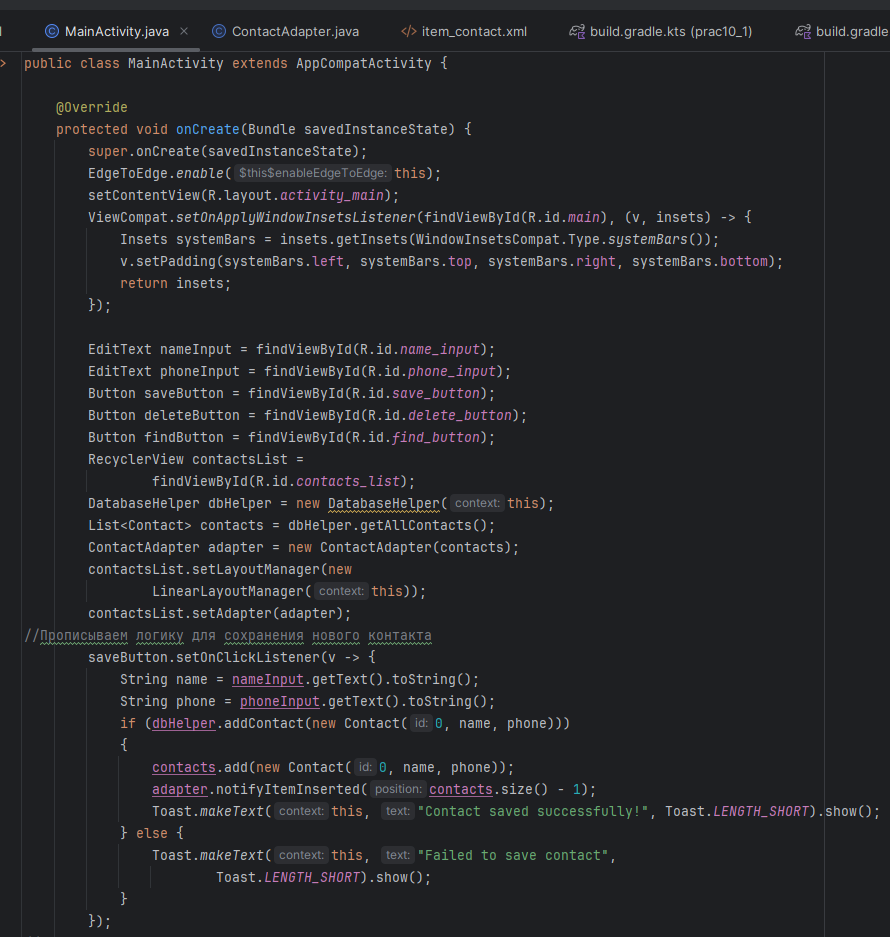


Рисунок – Описание класса MainActivity, часть 1

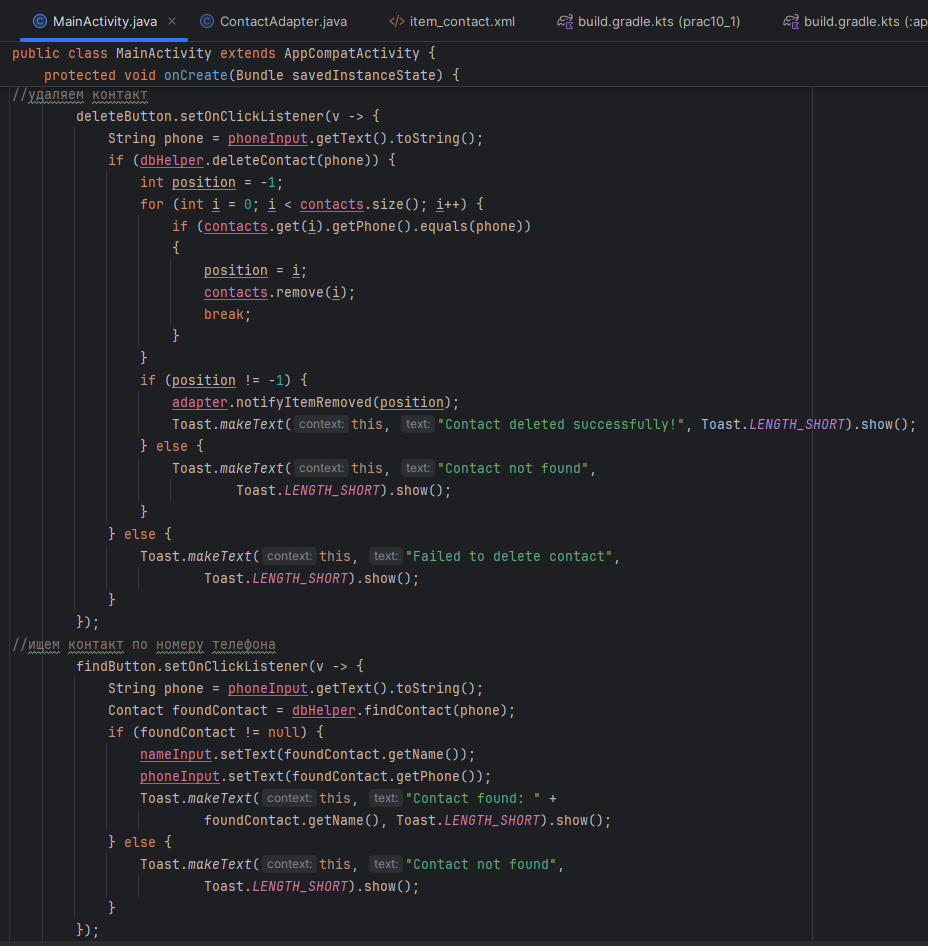


Рисунок – Описание класса MainActivity, часть 2

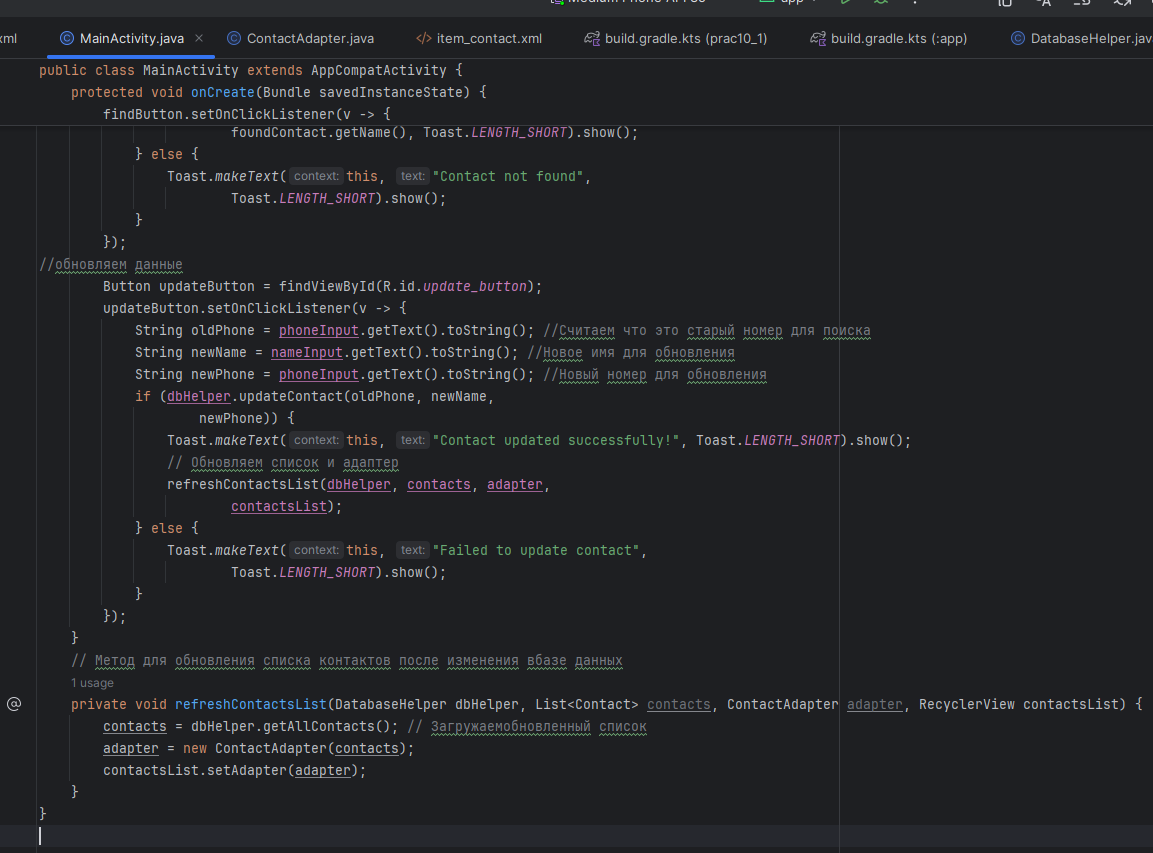


Рисунок – Описание класса MainActivity, часть 3

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

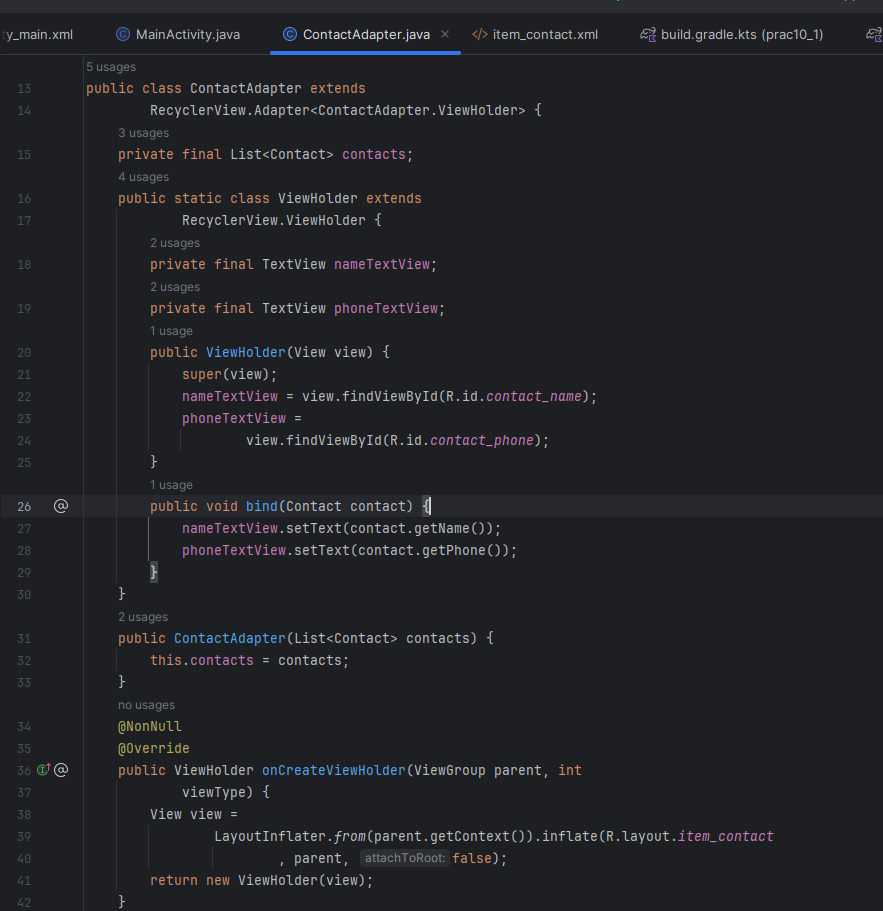


Рисунок – Описание класса ContactAdapter, часть 1

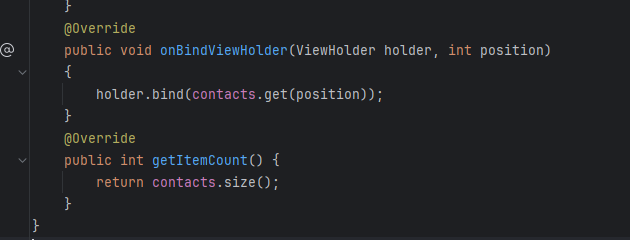


Рисунок – Описание класса ContactAdapter, часть 2

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

# ПРАКТИЧЕСКОЕ ЗАДАНИЕ

ЗАКЛЮЧЕНИЕ

В ходе данной практической работы были получены знания по обработке файлов во внешнем и внутреннем хранилищах, а также способ сохранения состояния приложение при смене его ориентации. Полученные знания были закреплены путём выполнения практического задания.