МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Практическое занятие № 6/2ч.

Разработка мобильных приложений

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень: бакалавриат
(бакалавриат, магистратура, специалитет)

Форма обучения: очная
(очная, очно-заочная, заочная)

Направление(-я) подготовки: 09.03.02 «Информационные системы и технологии»
(код(-ы) и наименование(-я))

Институт: кибербезопасности и цифровых технологий
(полное и краткое наименование)

Кафедра: КБ-14 «Цифровые технологии обработки данных»
(полное и краткое наименование кафедры, реализующей дисциплину (модуль))

Используются в данной редакции с учебного года 2024/25
(учебный год цифрами)

Проверено и согласовано «____» _____20___г. _____ _____
(подпись директора Института/Филиала с расшифровкой)

Москва 2025 г.

ОГЛАВЛЕНИЕ

1 ХРАНЕНИЕ ДАННЫХ В OS ANDROID

В OS «*Android*» существует несколько способов хранения данных, представленных на рисунке 1.1. Имеется возможность прямого доступа к внутренним и внешним областям хранения. Платформа «*Android*» поддерживает работу с базой данных «*SQLite*» для хранения реляционных данных, взаимодействие со специальными файлами для хранения пар «ключ-значение», а также использование сторонних нереляционных типов СУБД.

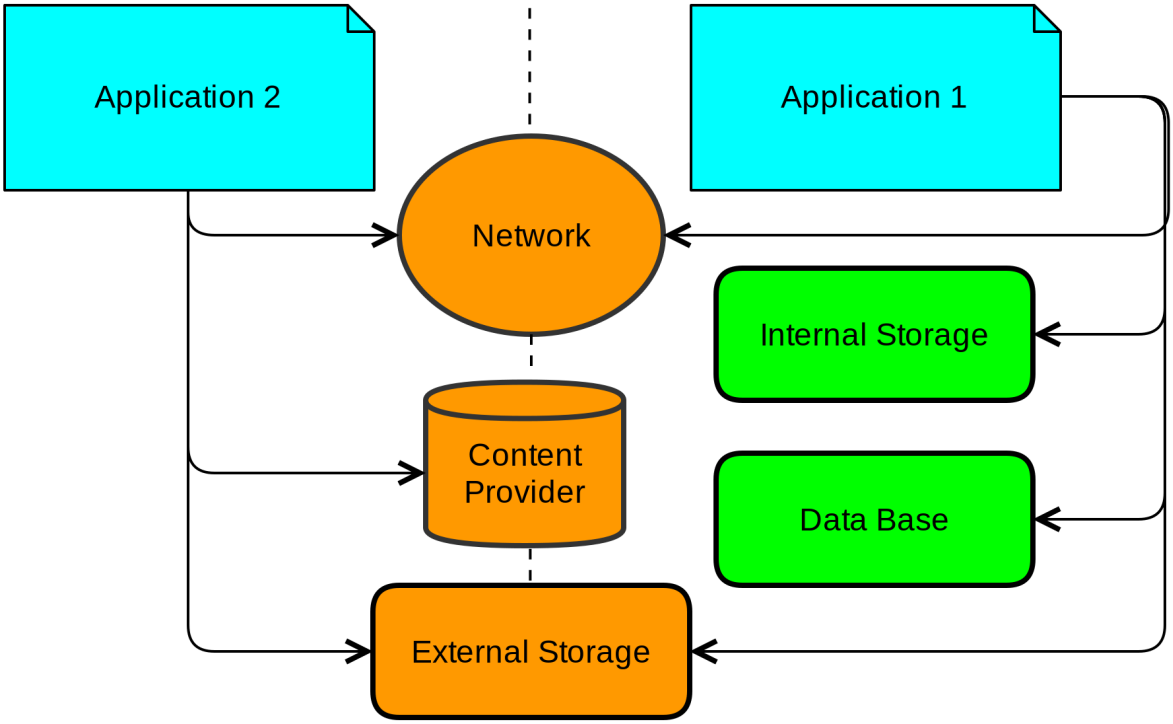

Рисунок 1.1 – Способы хранения данных

2 SHARED PREFERENCES

Основным способом хранения нескольких строк или номеров является специальный файл настроек (от англ. *preferences*). Хранение данных осуществляется в виде «ключ-значение» и используется для хранения глобальных данных. Также возможно хранить небольшие структуры, предварительно конвертированные в «JSON» и преобразованные в строковый тип данных. Для получения экземпляра

класса «*SharedPreferences*» и доступа к файлам настройки в коде приложения используются три метода:

- «*getPreferences*» – используется при одном общем файле настроек для приложения;
- «*getSharedPreferences*» – применяется при наличии нескольких общих файлов настроек с различным именем;
- «*getDefaultSharedPreferences*» – указывается при использовании файла по умолчанию.

Активности и службы могут использовать любой из представленных методов класса «*PreferenceManager*» для обеспечения возможности чтения и для записи в файл настроек.

```
SharedPreferences sharedPref =
                    getSharedPreferences("mirea_settings", Context.MODE_PRIVATE);
```

Константа «*MODE_PRIVATE*» используется для настройки ограничения доступа к файлу настроек только родительскому приложению. Запись в файл настроек выполняется с помощью вызова метода «*edit*» объекта «*SharedPreferences*».

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

Объект «*SharedPreferences.Editor*» имеет несколько методов, предназначенных для записи пар «ключ-значение» в файл настроек. Например, метод «*putString*» используется для хранения значения типа «*String*». После того, как произведено добавление пар ключ-значения, требуется вызов метода «*apply*» объекта «*SharedPreferences.Editor*» для их сохранения.

```
editor.putString("GROUP", "XXXX-XX-XX");
editor.putInt("NUMBER", 25);
editor.putBoolean("IS_EXCELLENT", true);

editor.apply();
```

Для получения сохраненных значений по ключу используется класс «*SharedPreferences*» с помощью методов:

- *getBoolean(String key, boolean defValue)*;
- *getFloat(String key, float defValue)*;
- *getInt(String key, int defValue)*;

- *getLong(String key, long defValue)*;
- *getString(String key, String defValue)*.

Далее представлен фрагмент кода, извлекающий записанное значение по установленному ключу.

```
String name = preferences.getString("GROUP ", "unknown");
int age = preferences.getInt("NUMBER ", 0);
boolean isSingle = preferences.getBoolean("IS_EXCELLENT ", false);
```

Вторым параметром методов «*get**» является возвращаемое по умолчанию значение в случае его отсутствия по ключу. Стоит обратить внимание, что возможные типы хранимой информации в файлах настроек ограничены только строками и примитивными типами данных. Файлы настроек хранятся в каталоге «*/data/data/имя_пакета/shared_prefs/имя_файла_настроек.xml*». Просмотр файлов настройки возможно осуществить с помощью файлового менеджера устройства среды разработки: «*View | Tool Windows | Device Explorer*».

2.1 Задание

Требуется создать новый проект *«ru.mirea.«фамилия».Lesson6».*

На экране необходимо разместить три поля ввода, и кнопку для сохранения информации. Требуется запомнить номер группы, номер по списку и название любимого фильма или сериала с помощью «*getSharedPreferences*». После новой загрузки приложения в поле ввода должны отобразиться значения из памяти. Для ВЫПОЛНЕНИЯ ЗАДАНИЯ требуется:

- найти файл настроек с помощью «*Device Explorer*» в среде разработки «*Android Studio*»;
- открыть файл настроек;
- создать директорию «*raw*» в проекте;
- сделать скриншот экрана со значениями файла и разместить его в директории «*raw*».

2.2 Задание

Создать новый модуль. В меню «*File | New | New Module | Phone & Tablet Module | Empty Views Activity*». Название модуля «*SecureSharedPreferences*». Создать экран

отображения имени Вашего любимого поэта и его фотографии или рисунка.

Основным недостатком использования «*SharedPreferences*» является то, что с помощью данного механизма данные хранятся в открытом виде. Для повышения уровня безопасности хранимых данных используется класс «*EncryptedSharedPreferences»* из библиотеки «*security-crypto*» в составе компонентов «*Jetpack».* В первую очередь требуется добавить библиотеку в «*gradle*»-файл модуля.

```
dependencies {
    implementation("androidx.security:security-crypto:1.0.0")
    // ...
}
```

В текущей версии библиотеки содержится три основных части: «*MasterKeys», «EncryptedSharedPreferences*» и «*EncryptedFile*». Класс «*MasterKeys*» имеет один публичный метод «*getOrCreate*» и позволяет создавать мастер-ключ. Единственным параметром метода является экземпляр класса «*KeyGenParameterSpec*», позволяющий задавать алгоритм, режим шифрования, выравнивание и размер ключа. Алгоритмов шифрования по умолчанию является «*MasterKeys.AES256_GCM_SPEC*».

```
KeyGenParameterSpec keyGenParameterSpec = MasterKeys.AES256_GCM_SPEC;
String mainKeyAlias = MasterKeys.getOrCreate(keyGenParameterSpec);
```

Созданный ключ размещается в контейнере «*AndroidKeyStore»*, в котором хранятся криптографические ключи в доверенной среде исполнения («*Trusted Execution Environment*») или «*StrongBox*», что затрудняет их извлечение. Мобильные устройства начиная с ОС «*Android* 9» (уровень API 28) поддерживают механизм безопасности «*StrongBox Keymaster*», который находится в аппаратном модуле безопасности.

Класс «*EncryptedSharedPreferences»* является дочерним классом «*SharedPreferences»*. Позволяет сохранять и читать значения, выполнять шифрование и расшифровывание данных внутри его реализации. Для вызова конструктора класса требуется вызвать метод «*create*».

```
SharedPreferences secureSharedPreferences = EncryptedSharedPreferences.create(
      "secret_shared_prefs",
      mainKeyAlias,
      getBaseContext(),
      EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
      EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
);

secureSharedPreferences.edit().putString("secure", "ЛЮБИМЫЙ АКТЕР")
```

Первым параметром является имя файла настроек, а вторым – псевдоним ключа для мастер-ключа. Последние два параметра задают схемы, используемые для шифрования ключей и их значения. На текущий момент ключи шифруются с использованием AES256-SIV-CMAC, который обеспечивает детерминированный шифрованный текст; значения зашифрованы с помощью AES256-GCM и привязаны к зашифрованному ключу. Запись данных осуществляется с помощью вызова методов «*put**».

```
try {
   String mainKeyAlias ….
   SharedPreferences secureSharedPreferences  …..
   secureSharedPreferences.edit().putString("secure", "ЛЮБИМЫЙ АКТЕР").apply();
} catch (GeneralSecurityException | IOException e) {
   throw new RuntimeException(e);
}
```

Получение данных осуществляется аналогичным с «*SharedPreferences*» способом.

```
String result = secureSharedPreferences.getString("secure", "ЛЮБИМЫЙ АКТЕР");
```

Для ВЫПОЛНЕНИЯ ЗАДАНИЯ требуется:

– найти файл настроек с помощью «*Device Explorer*» в среде разработки «*Android Studio*»;

– открыть файл настроек;

– создать директорию «*raw*» в проекте;

– сделать скриншот экрана со значениями файла и разместить его в директории «*raw*».

3 РАБОТА С ФАЙЛАМИ.

3.1 Запись файлов во внутреннее хранилище

Создать новый модуль. В меню «*File* | *New* | *New Module* | *Phone & Tablet Module* | *Empty Views Activity*». Название модуля «*InternalFileStorage*».

Работа с файлами настроек позволяет сохранить небольшие данные отдельных типов, но для работы с большими массивами данных, такими как графические файлы, файлы мультимедиа и т.д. требуется их хранение в файловой системе. Для разграничения между директориями в ОС «*Linux»* использует символ в виде косой черты, направленной слева направо «/», а не справа налево «\» (как в ОС «*Windows*»). Приложение для хранения данных в приватной директории используют собственный каталог «*/data/data/package_name/*». По умолчанию, все размещенные файлы доступны только тому приложению, которое их создало. Разумеется, возможно создание доступных файлов для других приложений. В случаях, когда приложение не предоставляет доступ к файлам извне, получить доступ к ним возможно только с правами «root»-пользователя.

Для работы с файлами в абстрактном классе «*android.content.Context*» определяется ряд методов:

- «*deleteFile*» – удаление определенного файла;
- «*fileList*» – получение всех файлов, содержащихся в подкаталоге *«files»* в директории приложения;
- «*getCacheDir*» – получение ссылки на подкаталог «*cache»* в директории приложения;
- «*getDir*» – получение ссылки на подкаталог в директории приложения, а в случае его отсутствия создание директории;
- «*getExternalCacheDir*» – получение ссылки на папку *«cache»* внешней файловой системы устройства;
- «*getExternalFilesDir*» – получение ссылки на каталог *«files»* внешней файловой системы устройства;
- «*getFileStreamPath»* – возвращение абсолютного пути к файлу в файловой системе;

– «*openFileInput*» – открытие файла для чтения;

– «*openFileOutput*» – открытие файла для записи.

Файлы, создаваемые и редактируемые в приложении, как правило, хранятся в подкаталоге «*files*» в директории приложения. Для непосредственного чтения и записи файлов применяются также стандартные классы «*Java*» из пакета «*java.io*». Далее представлен пример реализации записи информации в файл с помощью класса «*FileOutputStream*».

```
public class MainActivity extends AppCompatActivity {
    private static final String LOG_TAG = MainActivity.class.getSimpleName();
    private String fileName = "mirea.txt";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        String string = "Hello mirea!";
        FileOutputStream outputStream;
        try {
            outputStream = openFileOutput(fileName, Context.MODE_PRIVATE);
            outputStream.write(string.getBytes());
            outputStream.close();
        } catch (Exception e) {
            e.printStackTrace();
        }}
}
```

Система позволяет создавать файлы с двумя разными режимами:

– «*MODE_PRIVATE*» – файлы доступны только владельцу приложения (режим по умолчанию);

– «*MODE_WORLD_READABLE*» – файл доступен для чтения всем;

– «*MODE_WORLD_WRITEABLE*» – файл доступен для записи всем (считается устаревшим с API 17);

– «*MODE_APPEND*» – данные могут быть добавлены в конец файла.

Для просмотра созданных файлов на эмуляторе требуется открыть экран: «*View|Tool Windows|Device Explorer*», представленный на рисунке 3.1.

Рисунок 3.1 – Способы хранения данных

Созданный файл размещается в директории «*/data/data/package_name/files*» (рисунок 3.2).

Рисунок 3.2 – Способы хранения данных

Для чтения файла применяется поток ввода, реализованный в классе «*InputStream*» и позволяющий указывать различные источники:

- массив байтов;
- строка («*String*»);
- файл;
- канал («*pipe*») – однонаправленный канал межпроцессного взаимодействия, в котором данные передаются с одного конца и извлекаются в другом;
- последовательность различных потоков, которые возможно объединить в одном потоке;
- другие источники (например, подключение к интернету).

Для работы с указанными источниками используются подклассы базового класса «*InputStream*»:

- «*BufferedInputStream*» – буферизированный входной поток;
- «*ByteArrayInputStream*» – буфер в памяти (массив байтов);
- «*DataInputStream*» – входной поток, включающий методы для чтения стандартных типов данных «*Java*»;
- «*FileInputStream*» – чтение информации из файла;
- «*FilterInputStream*» – интерфейс для классов-надстроек, которые добавляют к существующим потокам различные свойства;
- «*InputStream*» – абстрактный класс, описывающий поток ввода;
- «*ObjectInputStream*» – входной поток для объектов;
- «*StringBufferInputStream*» – превращает строку во входной поток данных

«*InputStream*»;

- «*PipedInputStream»* – реализует понятие входного канала;
- «*PushbackInputStream»* – входной поток, поддерживающий однобайтовый возврат во входной поток;
- «*SequenceInputStream»* – объединение двух или более потока «*InputStream»* в единый поток.

К основным методам класса «*InputStream*» относятся:

- «*available*» – возвращает количество байтов ввода, доступных в данный момент для чтения;
- «*close*» – закрывает источник ввода. Следующие попытки чтения вернут исключение «*IOException»*;
- «*mark(int readlimit)*» – помещает метку в текущую точку входного потока, которая остаётся корректной до тех пор, пока не будет прочитано «*readlimint*» байт;
- «*markSupported*» – возвращает «*true»*, если методы «*mark*» и «*reset*» поддерживаются потоком;
- «*read*» – возвращает целочисленное представление следующего доступного байта в потоке. При достижении конца файла возвращается значение «*-1*».
- «*read(byte[] buffer)*» – считывание байтов в буфер, возвращая количество прочитанных байтов. По достижении конца файла возвращается значение «*-1*»;
- «*int read(byte[] buffer, int byteOffset, int byteCount)*» – считывание количества байт в соответствии с «*byteCount»* начиная со смещения «*byteOffset»*. По достижении конца файла возвращается значение «*-1*»;
- «*reset*» – выполняется сброс входного указателя в ранее установленную метку;
- «*long skip(long byteCount)*» – пропускает определенное количество «*byteCount»*, возвращая количество проигнорированных байтов.

В представленном методе «*onCreate*» производится вызов метода «*getTextFromFile*», который считывает файл и возвращает строку. Считывание

данных и установка значений производится в новом потоке (см. 4 практическое занятие).

```
@Override
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      tv = findViewById(R.id.textView);
      ...
      new Thread(new Runnable() {
          public void run() {
              try {
                  Thread.sleep(5000);
              } catch (InterruptedException e) {
                  e.printStackTrace();
              }
              tv.post(new Runnable() {
                  public void run() {
                      tv.setText(getTextFromFile());
                  }
              });
          }
      }).start();
  }
  // открытие файла
  public String getTextFromFile() {
      FileInputStream fin = null;
      try {
          fin = openFileInput(fileName);
          byte[] bytes = new byte[fin.available()];
          fin.read(bytes);
          String text = new String(bytes);
          Log.d(LOG_TAG, text);
          return text;
      } catch (IOException ex) {
          Toast.makeText(this, ex.getMessage(), Toast.LENGTH_SHORT).show();
      } finally {
          try {
              if (fin != null)
                  fin.close();
          } catch (IOException ex) {
              Toast.makeText(this, ex.getMessage(), Toast.LENGTH_SHORT).show();
          }
      }
      return null;
  }
}
```

Для ВЫПОЛНЕНИЯ ЗАДАНИЯ требуется:

– добавить на экран поле ввода и кнопку;

– записать в файл памятную дату в истории России и ее описание;

– создать директорию «*raw*» в проекте;

– переместить созданный файл с эмулятора или устройства в проект.

3.2 Запись файлов во внешнее хранилище

В мобильных устройствах возможно производить запись данных во внешнее хранилище. Однако, стоит учитывать, что память может быть недоступна –

например при подключении устройства к компьютеру или при удалении SD карты. Требуется всегда проверять раздел на доступность, прежде чем его использовать. Запрос состояния внешнего хранилища осуществляется с помощью вызова метода «*getExternalStorageState*». В случаях, когда метод вернул состояние, равное «*MEDIA_MOUNTED*», возможно читать и записывать файлы. Пример метода проверки внешнего хранилища на доступность:

```
/* Проверяем хранилище на доступность чтения и записи*/
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}

/* Проверяем внешнее хранилище на доступность чтения */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
            Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

Для записи данных на внешнюю карту памяти требуется указать в манифесте разрешение и произвести запрос к пользователю на запись данных.

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
  tools:ignore="ScopedStorage" />
```

На примере класса «*OutputStreamWritter*» выполняется запись данных в файл в общем каталоге «*Documents*»:

```
public void writeFileToExternalStorage() {
  File path = Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOCUMENTS);
  File file = new File(path, "FAVORITE_QUOTE.txt");
  try {
    FileOutputStream fileOutputStream = new FileOutputStream(file.getAbsoluteFile());
    OutputStreamWriter output = new OutputStreamWriter(fileOutputStream);
    // Запись строки в файл
    output.write("data");
    // Закрытие потока записи
    output.close();

  } catch (IOException e) {
    Log.w("ExternalStorage", "Error writing " + file, e);
  }
}
```

3.3 Задание

Создать новый модуль. В меню «*File | New | New Module | Phone & Tablet Module | Empty Views Activity*». Название модуля «Notebook».

Требуется создать приложение – «Блокнот» с сохранением файлов.

Для ВЫПОЛНЕНИЯ ЗАДАНИЯ требуется:

– добавить на экран поля ввода «названия файла» и «цитата»;

– добавить на экран кнопки «сохранить данные в файл» и «загрузить данные из файла»;

– файлы сохраняются в публичную директорию «*Directory_Documents*» с перезаписью;

– при загрузке файла считываются данные и устанавливаются в поле «цитата»;

– требуется записать в два файла цитаты известных людей;

– создать директорию «*raw*» в проекте;

– переместить созданные файлы с эмулятора или устройства в проект.

Пример чтения файла:

```
public void readFileFromExternalStorage() {
   File path = Environment.getExternalStoragePublicDirectory(
         Environment.DIRECTORY_DOCUMENTS);
   File file = new File(path, "FAVORITE_QUOTE.txt");
   try {
      FileInputStream fileInputStream = new FileInputStream(file.getAbsoluteFile());
      InputStreamReader inputStreamReader = new InputStreamReader(fileInputStream, Standard-
Charsets.UTF_8);
      List<String> lines = new ArrayList<String>();
      BufferedReader reader = new BufferedReader(inputStreamReader);
      String line = reader.readLine();
      while (line != null) {
         lines.add(line);
         line = reader.readLine();
      }
      Log.w("ExternalStorage", String.format("Read from file %s successful", lines.toString()));

   } catch (Exception e) {
      Log.w("ExternalStorage", String.format("Read from file %s failed", e.getMessage()));
   }
}
```

4 БАЗА ДАННЫХ SQLITE

4.1 SQLite

На мобильных устройствах используется компактная встраиваемая СУБД «*SQLite*». Проект с открытым исходным кодом, поддерживающим стандартные возможности обычной SQL: синтаксис, транзакции и др. Объем занимаемого места составляет около 250 кб. и поддерживает несколько типов данных, представленных в таблице 4.1. Остальные типы требуют конвертации, прежде чем их сохранять в базе данных. «*SQLite*» не проверяет типы данных, поэтому разработчик может записать целое число в колонку, предназначенную для строк, и наоборот.

Таблица 4.1 Поддерживаемые типы данных в SQLite

Тип	**Описание**
NULL	пустое значение
INTEGER	целочисленное значение
REAL	значение с плавающей точкой
TEXT	строки или символы в кодировке UTF-8, UTF-16BE или UTF-16LE

Стоит отметить, что отсутствует тип данных, предназначенный для хранения даты. Решением является использование строковых значений, например «2023-04-20». Для даты со временем рекомендуется использовать формат «2023-04-20T09:10». В таких случаях возможно использование некоторых функций «*SQLite*» для добавления дней, установки начала месяца и т.д. Отсутствие поддержки логического типа данных решается использованием числа значений «0» для «ЛОЖЬ» и 1 – «ИСТИНА». Для хранения файлов (напр. медиа данные) лучшим решением будет использование пути в базе, а сами изображения хранить в файловой системе.

Начальным этапом работы с базой данных является установка необходимых настроек для её создания или обновления. База данных «*SQLite*» представляет собой файл, поэтому операции чтения и записи могут быть довольно медленными. Рекомендуется использовать асинхронные операции.

При запуске приложения создаётся база данных в каталоге

«*/data/имя_пакета/databases/имя_базы.db*». Основными пакетами для работы с базой данных являются «*android.database*» и «*android.database.sqlite*».

База данных «*SQLite*» доступна только приложению, которое создаёт её. Если требуется предоставить доступ к данным другим приложениям, используется контент-провайдеры («*ContentProvider*»).

4.2 Классы для работы с SQLite

Работа с базой данных «*SQLite*» заключается в:

- создании и открытии базы данных;
- создании таблиц;
- создании интерфейса для вставки данных;
- создании интерфейса для выполнения запросов (выборка данных);
- закрытии базы данных.

На рисунке 4.1 представлена структура механизма взаимодействия активности с БД.

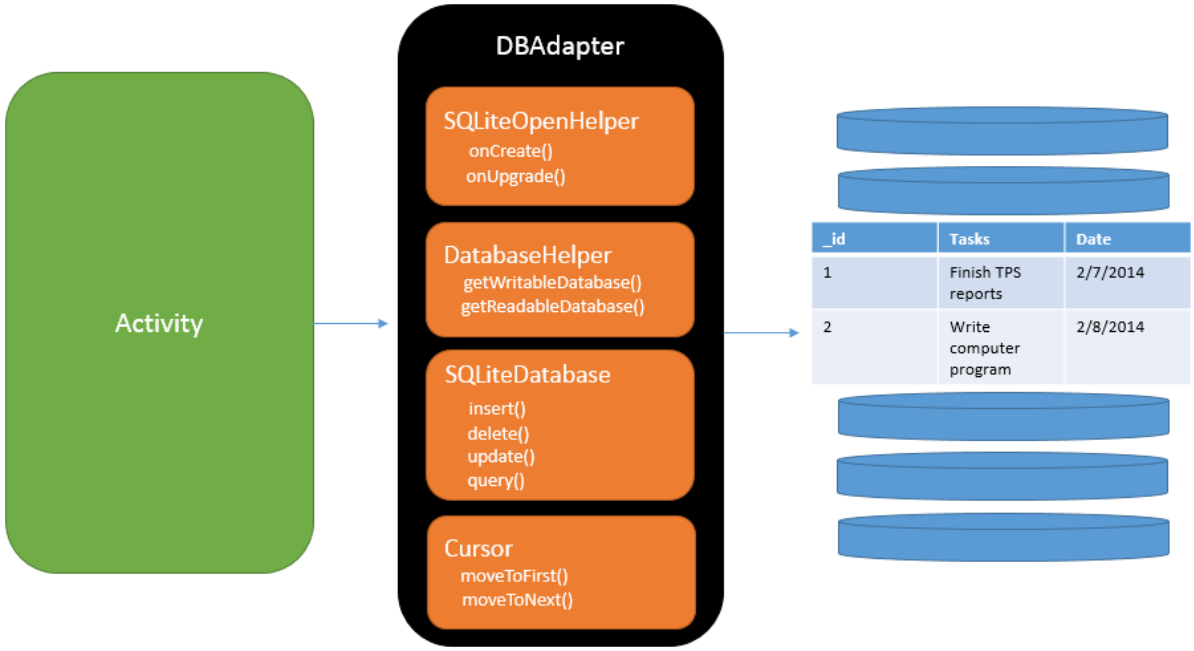

Рисунок 4.1 - Структура механизма взаимодействия приложения с БД

Для добавления новых строк в таблицу используется класс «*ContentValues*». Каждый объект данного класса представляет собой одну строку таблицы и имеет вид ассоциативного массива с именами столбцов и значениями, которые им соответствуют.

В ОС «*Android*» запросы к базе данных возвращают объекты класса «*Cursor*». Вместо извлечения данных и возвращения копий значений, курсоры ссылаются на результирующий набор исходных данных. Курсоры позволяют управлять текущей позицией (строкой) в результирующем наборе данных, возвращаемом при запросе.

Абстрактный класс «*SQLiteOpenHelper*» предназначен для создания, открытия и обновления базы данных. Данный класс является основным классом. При реализации данного вспомогательного класса от разработчика скрывается логика, на основе которой принимается решение о создании или обновлении базы данных перед ее открытием. Класс «*SQLiteOpenHelper*» содержит два обязательных абстрактных метода:

- «*onCreate*» – вызывается при первом создании базы данных;
- «*onUpgrade*» – вызывается при модификации базы данных.

Также используются другие методы класса:

- «*onDowngrade(SQLiteDatabase, int, int)*» – вызывается при понижении версии базы данных;
- «*onOpen(SQLiteDatabase)*» – вызывается при открытии базы данных.
- «*getReadableDatabase()*» – создает или открывает базу данных для чтения;
- «*getWritableDatabase()*» – создает или открывает базу данных для записи;

Для управления базой данных «*SQLite»* применяется класс «*SQLiteDatabase*». В классе «*SQLiteDatabase»* определены методы «*query», «insert», «delete»* и *«update»* для чтения, добавления, удаления, изменения данных. Кроме того, метод «*execSQL*» позволяет выполнять любой допустимый код на языке SQL применимо к таблицам базы данных. После редактирования значений в базе данных необходимо выполнять вызов метода «*refreshQuery*» для всех курсоров, которые имеют отношение к редактируемой таблице. Для составления запроса используются два метода: «*rawQuery*» и «*query*», а также класс «*SQLiteQueryBuilder*».

```
Cursor query (String table,
              String[] columns,
              String selection,
              String[] selectionArgs,
              String groupBy,
              String having,
              String sortOrder)
```

В метод «*query»* возможно передать семь параметров:

- «*table*» – имя таблицы, к которой будет передан запрос;
- «*String[] columnNames*» – список имен возвращаемых полей (массив). При передаче «*null»* возвращаются все столбцы;
- «*String whereClause*» – параметр, формирующий выражение «*WHERE*». Значение «*null»* возвращает все строки*;*
- «*String[] selectionArgs*» – значения аргументов фильтра;
- «*String[] groupBy*» – фильтр для группировки, формирующий выражение «*GROUP BY*»;
- «*String[] having*» – фильтр для группировки, формирующий выражение «*HAVING*»;
- «*String[] orderBy*» – параметр сортировки, формирующий выражение ORDER BY.

4.3 Data Access Object

При работе с базами данных выделяются слои, отвечающие за взаимодействие между различными модулями системы. Отдельным слоем является модуль, отвечающий за передачу запросов в БД и обработку полученных от неё ответов. Шаблон «*Data Access Object*» (DAO) является структурным шаблоном, позволяющий выполнить изоляцию прикладного/бизнес-уровня от постоянного уровня (обычно это реляционная база данных, но это может быть любой другой постоянный механизм) с использованием абстрактного API.

Функциональность данного API заключается в реализации программного кода, связанного с выполнением операций CRUD в базовом механизме хранения, что позволяет слоям развиваться раздельно.

На рисунке 4.2 представлена архитектура паттерна DAO.

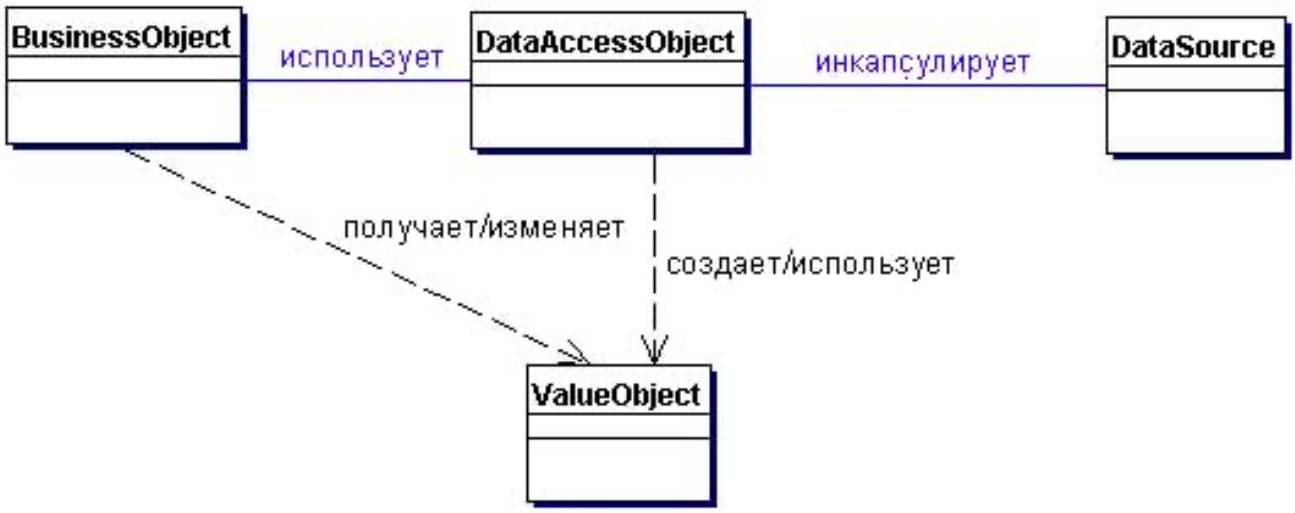

Рисунок 4.2 - Архитектура паттерна DAO

Данный паттерн состоит из следующих объектов:

– «*BusinessObject*» представляет клиента с доступом к источнику данных для их получения и сохранения;

– «*DataAccessObject» (далее* DAO*)* является первичным объектом данного паттерна, позволяющий абстрагировать реализацию доступа к данным для «*BusinessObject*» и обеспечивающий прозрачный доступ к источнику данных. «*BusinessObject*» передает также ответственность за выполнение операций загрузки и сохранения данных объекту DAO;

– «*DataSource*» представляет реализацию источника данных, которым может быть база данных, XML документы, данные в формате JSON и другие;

– «*ValueObject»* представляет собой объект, используемый для передачи данных. DAO может использовать его для возврата и приема данных клиенту.

На рисунке 4.3 представлена диаграмма последовательности действий, показывающая взаимодействия между различными сущностями в данном паттерне.

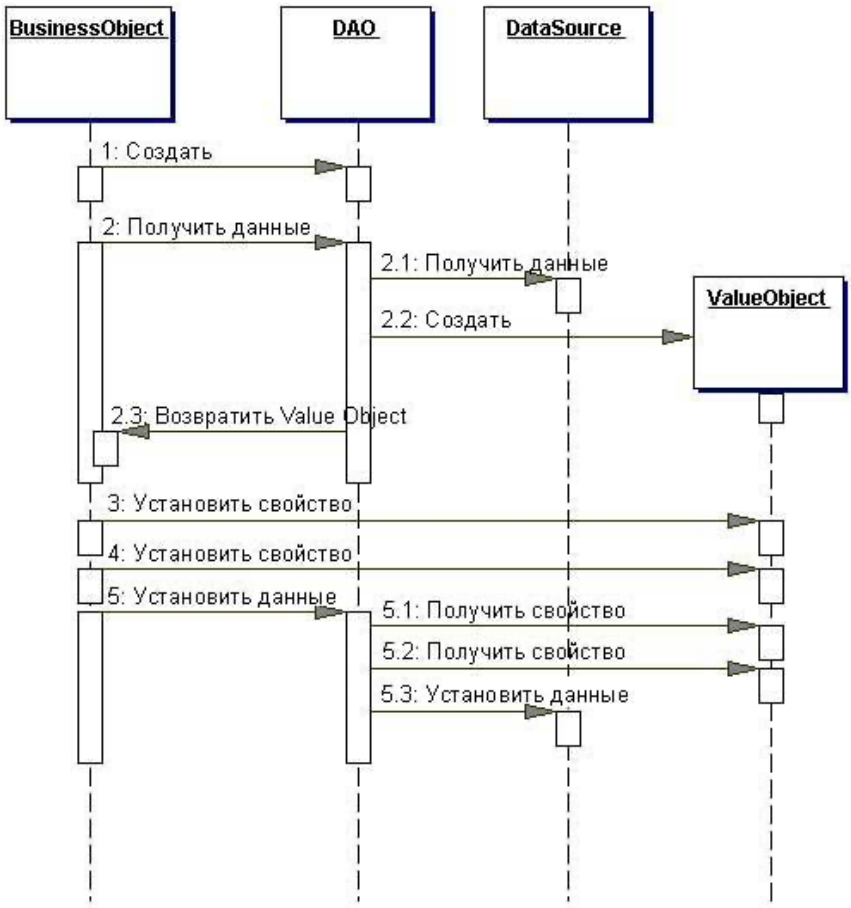

Рисунок 4.3 – Диаграмма последовательности действий паттерна Data Access Object

4.4 Room

Работа с базой данных «*SQLite*» в ОС «*Android*» не отличается удобством из-за большого объема кодовой базы. Для решения задач хранения данных на мобильных устройствах имеется несколько наиболее популярных вариантов от сторонних разработчиков: «*Realm», «ORMLite», «GreenDao», «DBFlow», «Sugar ORM*». Для облегчения задач работы с БД в 2017 году корпорацией «*Google*» была разработана библиотека «*Room*», представляющая собой обертку для работы с базой данных «*SQLite*», входящую в набор библиотек под названием «*Android Architecture Components*».

Компонент «*Room*» является обёрткой для класса «*SQLiteOpenHelper*», т.е. ORM (англ. *Object-relational mapping*) между классами «*Java*» и «*SQLite*». Компонент возможно разделить на три части: «*Entity»*, «DAO (Data Access Object)» и «Database». Сущность «*Entity»* является объектным представлением таблицы, в

которой с помощью аннотаций возможно описать поля. Для создания «*Entity*» требуется создать класс «*Plain Old Java Object*» (POJO) и пометить класс аннотацией «*Entity*».

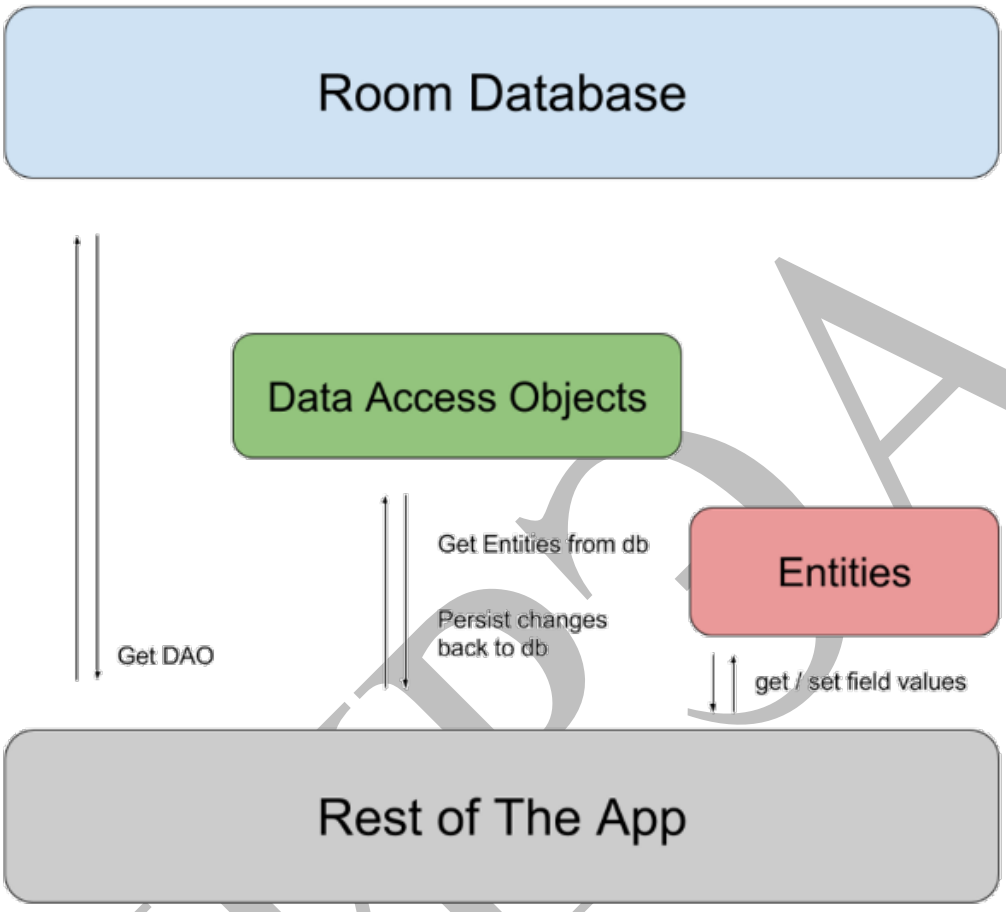

Рисунок 4.4 – Структура компонента «Room»

4.5 Задание

Создать новый модуль. В меню «*File | New | New Module | Phone & Tablet Module | Empty Views Activity*». Название модуля «EmployeeDB». Создать базу данных для хранения информации о вымышленных супер-героях.

Компонент «*room*» не входит в стандартный пакет «*Android SDK*» и для его использования требуется в файл модуля «*build.gradle»* добавить зависимость в блок «*dependencies*» и синхронизировать проект:

```
dependencies {
    implementation("androidx.room:room-runtime:2.6.1")
    annotationProcessor("androidx.room:room-compiler:2.6.1")
    // ...
}
```

На первом шаге необходимо создать класс, на основании которого будет

создана таблица базы данных. Используя аннотации над классами компонент «*Room*» генерирует таблицы базы данных. Аннотация «*@Entity*» способствует созданию таблицы с именем, совпадающим с именем класса. Каждый класс с аннотацией «*@Entity*» обязательно должен обладать первичным ключом, отмеченным аннотацией «*@PrimaryKey*». Возможно добавление дополнительных атрибутов для генерации уникального идентификатора «*autoGenerate*» (аналог ключевого слова «*AUTOINCREMENT*» в SQL), либо разработчику потребуется следить за уникальностью записи вручную. Поля таблицы создаются в соответствии с полями класса.

```
@Entity
public class Employee {
    @PrimaryKey(autoGenerate = true)
    public long id;
    public String name;
    public int salary;
}
```

В созданном классе имеется вся информация, необходимая для создания таблицы компонентом «*Room*».

На втором шаге требуется создание механизма взаимодействия с базой данных и управления данными (добавление, удаление, формирование запросов). Для решения задачи используется интерфейс DAO с помощью создания нового класса-интерфейса.

```
@Dao
public interface EmployeeDao {
    @Query("SELECT * FROM employee")
    List<Employee> getAll();
    @Query("SELECT * FROM employee WHERE id = :id")
    Employee getById(long id);
    @Insert
    void insert(Employee employee);
    @Update
    void update(Employee employee);
    @Delete
    void delete(Employee employee);
}
```

Методы «*getAll*» и «*getById*» предназначены для получения полного списка сотрудников или конкретного сотрудника по его идентификатору. Стоит обратить внимание, что в качестве имени таблицы используется имя «*employee*», соответствующее названию «*Entity*»-класса, т.е. «*Employee*», но без учета регистра в именах таблиц. Аннотация «*@Insert*» предназначена для вставки данных,

а с указанием аргумента «*onConflict = REPLACE*» при обнаружении записи с одинаковыми уникальными индексами производится замена содержимого. Аннотации «*@Update*» и «*@Delete*» обновляют и удаляют записи из таблицы. Аннотация «*@Query*» позволяет формировать запросы на языке SQL. Запросы в аннотациях проверяются на этапе компиляции приложения.

На третьем этапе производится создание абстрактного класса базы данных, наследуемого от «*RoomDatabase*» и отвечающего за ведение самой базы данных и предоставление экземпляров DAO. Данный класс должен содержать класс-модель, номер версии базы данных (изменяется при любой модификации модели данных), абстрактный метод без параметров, возвращающий аннотированный как «*@Dao*» класс.

```
@Database(entities = {Employee.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract EmployeeDao employeeDao();
}
```

Последним этапом является создание класса, наследуемого от класса «*Application*». Данный класс является реализацией паттерна «*Singleton*» и создаётся один раз при запуске приложения и предназначен для инициализации компонентов уровня приложения и хранения состояния. Создание такого класса требует вызова контекстного меню у директории, в которой размещён код приложения: «*File|New|Java/Kotlin Class*» и установки имени. Далее в manifest – файл необходимо добавить значение в блоке «*application*»:

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:name=".App"
        ...
```

В созданном классе производится инициализация базы данных в методе «*onCreate*». Необходимыми аргументами для создания объекта базы данных является «*ApplicationContext*», имя дочернего класса «*RoomDatabase*» – «*AppDatabase*» и название файла для базы.

```
public class App extends Application {
    public static App instance;
    private AppDatabase database;

    @Override
    public void onCreate() {
        super.onCreate();
        instance = this;
        database = Room.databaseBuilder(this, AppDatabase.class, "database")
                .allowMainThreadQueries()
                .build();
    }
    public static App getInstance() {
        return instance;
    }
    public AppDatabase getDatabase() {
        return database;
    }
}
```

При создании приложения производится инициализация базы данных. Вызов метода «*Room.build()*» каждый раз создает новый экземпляр базы. Экземпляры достаточно массивные и рекомендуется использовать один экземпляр для всех операций. Вызов функций работы с базой следует осуществлять в отдельном потоке, однако имеется возможность выполнения запросов в главном потоке для не ресурсоёмких задач с помощью метода «*allowMainThreadQueries()*». В качестве инструмента обмена данными между слоями, как правило, используется «*RxJava*» или компонент «*LiveData*» из «*Architecture Components*». Далее представлен пример работы с БД.

```
public class MainActivity extends AppCompatActivity {ы
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        AppDatabase db = App.getInstance().getDatabase();
        EmployeeDao employeeDao = db.employeeDao();

        Employee employee = new Employee();
        employee.id = 1;
        employee.name = "John Smith";
        employee.salary = 10000;
        // запись сотрудников в базу
        employeeDao.insert(employee);

        // Загрузка всех работников
        List<Employee> employees = employeeDao.getAll();
        // Получение определенного работника с id = 1
        employee = employeeDao.getById(1);
        // Обновление полей объекта
        employee.salary = 20000;
        employeeDao.update(employee);
        Log.d(TAG, employee.name + " " + employee.salary);
    }
}
```

5 КОНТРОЛЬНОЕ ЗАДАНИЕ

В контрольном задании «*MireaProject*» требуется:

– добавить фрагмент «Профиль», в котором пользователь должен указать определённые параметры (задумка исполнителя) и сохранить их в «*SharedPreferences*»;

– добавить фрагмент «Работа с файлами». Придумать функционал экрана, связанный с обработкой файлов (конвертация форматов, криптография, стеганография и т.д.). При нажатии на «*Floating Action Button*» вызывается диалоговое окно/фрагмент/View создания записи.