МИНОБРНАУКИ РОССИИ

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

РТУ МИРЭА

Практическое занятие № 6/4ч.

Разработка мобильных компонент анализа безопасности информационно-аналитических систем

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

Уровень: бакалавриат
(бакалавриат, магистратура, специалитет)

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

Направление(-я) подготовки: 10.05.04 Информационно-аналитические системы безопасности
(код(-ы) и наименование(-я))

Институт: комплексной безопасности и специального приборостроения ИКБСП
(полное и краткое наименование)

Кафедра: КБ-4 «Прикладные информационные технологии»
(полное и краткое наименование кафедры, реализующей дисциплину (модуль))

Используются в данной редакции с учебного года 2021/22
(учебный год цифрами)

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

Москва 2021 г.

ОГЛАВЛЕНИЕ

1 ХРАНЕНИЕ ДАННЫХ В OS ANDROID

В OS Android имеется несколько способов хранения данных (рисунок 1.1). Есть возможность прямого доступа к внутренним и внешним областям хранения Android-устройства, платформа Android предлагает базы данных SQLite для хранения реляционных данных, специальные файлы для хранения пар ключ-значение. Более того, приложения для Android также могут использовать сторонние базы данных, которые предлагают поддержку NoSQL. В 2017 году от компании Google была представлена библиотека Room, представляющая удобную обертку для работы с базой данных SQLite.

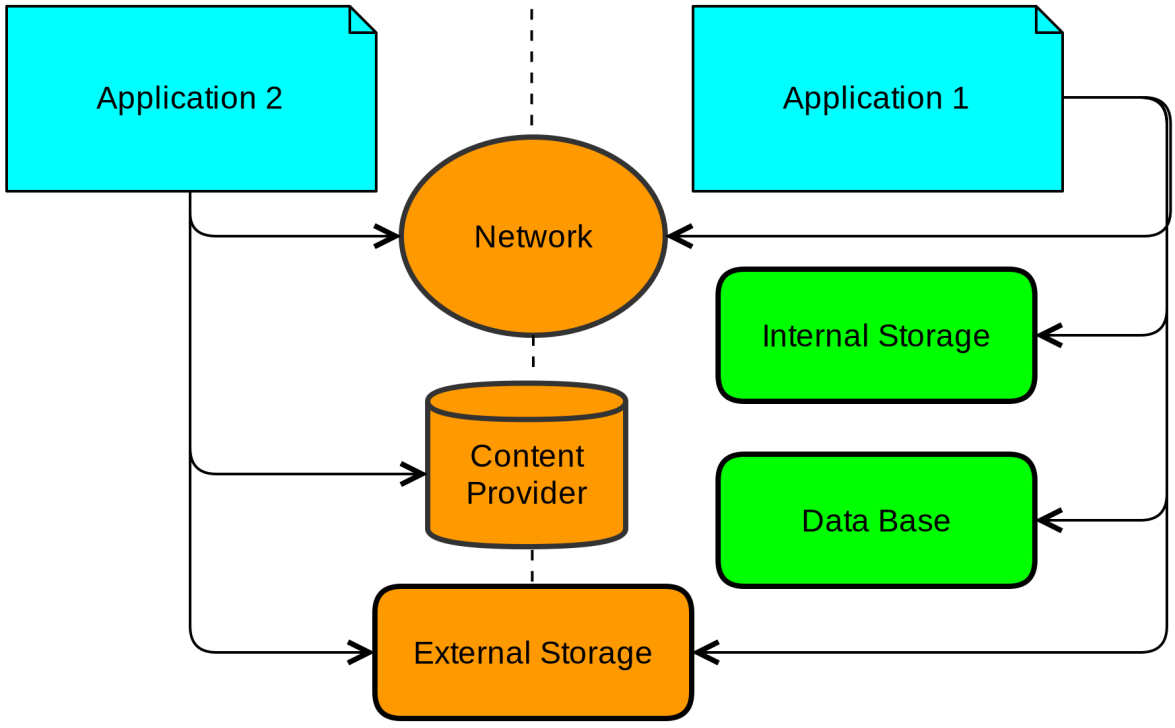


Рисунок 1.1 – Способы хранения данных

2 SHARED PREFERENCES

Для быстрого способа сохранения нескольких строк или номеров используется файл настроек (preferences). Хранение осуществляется в виде ключ-значение. Подходит для хранения глобальных данных. Также возможно хранить

небольшие структуры, предварительно конвертированные в JSON и преобразованные в String.

Для получения экземпляра класса SharedPreferences и доступа к настройкам в коде приложения используются три метода:

- getPreferences() — внутри активности, чтобы обратиться к определенному для активности предпочтению;
- getSharedPreferences() — внутри активности, чтобы обратиться к предпочтению на уровне приложения;
- getDefaultSharedPreferences() — из объекта PreferencesManager, чтобы получить общедоступную настройку, предоставляемую Android.

Активити и службы Android могут использовать метод getDefaultSharedPreferences() класса PreferenceManager, чтобы ссылаться на объект SharedPreferences, который может быть использован и для чтения, и для записи в файл настроек по умолчанию.

```
SharedPreferences preferences =
       getSharedPreferences(PERSISTANT_STORAGE_NAME, Context.MODE_PRIVATE);
```

Константа MODE_PRIVATE используется для настройки доступа и означает, что после сохранения, данные будут видны только родительскому приложению.

Чтобы начать запись в файл настроек, требуется вызвать метод *edit()* объекта SharedPreferences, который возвращает объект SharedPreferences.Editor.

```
SharedPreferences.Editor ed = preferences.edit();
```

Объект SharedPreferences.Editor имеет несколько интуитивных методов, которые возможно использовать для хранения новых пар ключ-значение в файле настроек. Например, метод putString() используется, чтобы поместить пару ключ-значение со значением типа String. Аналогично, возможно использовать метод putFloat(), чтобы поместить пару ключ-значение, чьё значение типа float. Все эти методы возвращают экземпляр класса SharedPreferences, из которого можно получить соответствующую настройку с помощью ряда 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).

Следующий пример кода создаёт три пары ключ-значение. После того, как добавлены все пары значений, требуется вызвать метод *apply()* объекта SharedPreferences.Editor, чтобы сохранить их.

```
editor.putString("NAME", "Alice");
editor.putInt("AGE", 25);
editor.putBoolean("SINGLE?", true);

editor.apply();
```

Чтение из объекта SharedPreferences производится вызовом соответствующего метода get*(). Например, чтобы получить пару ключ-значение, чьё значение является String, необходимо вызывать метод getString(). Вот фрагмент кода, который извлекает все значения, которые мы добавили ранее:

```
String name = preferences.getString("NAME", "unknown");
int age = preferences.getInt("AGE", 0);
boolean isSingle = preferences.getBoolean("SINGLE?", false);
```

Второй параметр всех методов get*() устанавливает значение по умолчанию в случае, если значения не существует. Обратите внимание, что файлы настроек ограничены только строками и примитивными типами данных.

Файлы настроек хранятся в каталоге */data/data/имя_пакета/shared_prefs/имя_файла_настроек.xml*. Поэтому в отладочных целях, если вам нужно сбросить настройки в эмуляторе, то при помощи перспективы DDMS, используя файловый менеджер, зайдите в нужную папку, удалите файл настроек и перезапустите эмулятор, так как эмулятор хранит данные в памяти, которые он сбрасывает в файл. На устройстве вы можете удалить программу и поставить ее заново, то же самое можно сделать и на эмуляторе, что бывает проще, чем удалять файл настроек вручную и перезапускать эмулятор.

2.1 Задание

Требуется создать новый проект *ru.mirea.«фамилия».practice6*. Название модуля Preferences.

Требуется запомнить имя пользователя и название учебного заведения. После загрузки приложения отображаются значения из памяти. На экране требуется

разместить – текстовые поля для отображения и ввода данных и 2 кнопки для сохранения и загрузки данных.

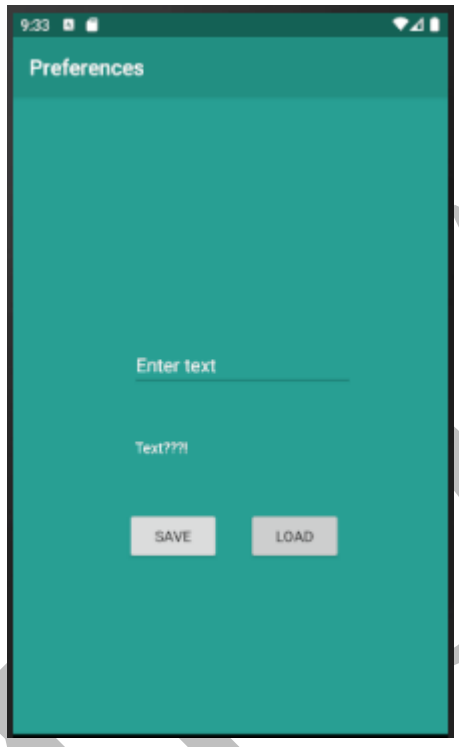


Рисунок 2.1 – Экран приложения Preferences

Далее рассматривается участок кода, приведённый ниже:

onSaveText – сохранение данных. Для того чтобы редактировать данные, необходим объект Editor – возвращаемый из объекта SharedPreferences. В методе putString указывается наименование переменной – это константа SAVED_TEXT, и значение – содержимое поля editText. Чтобы данные сохранились, необходимо вызвать функцию *apply*. Для наглядности отображается сообщение, что данные сохранены.

onLoadText – загрузка данных. Считывание данных производится с помощью метода getString – в параметрах указываем константу - это имя, и значение по умолчанию (пустая строка). Далее полученное значение устанавливается полю textView.

```
public class MainActivity extends AppCompatActivity {

    private EditText editText;
    private TextView textView;
    private SharedPreferences preferences;
    final String SAVED_TEXT = "saved_text";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        editText = findViewById(R.id.editText);
        textView = findViewById(R.id.textView);
        preferences = getPreferences(MODE_PRIVATE);
    }

    public void onSaveText(View view) {
        SharedPreferences.Editor editor = preferences.edit();
        // Сохранение значения по ключу SAVED_TEXT
        editor.putString(SAVED_TEXT, editText.getText().toString());
        editor.apply();

        Toast.makeText(this, "Text saved", Toast.LENGTH_SHORT).show();
    }

    public void onLoadText(View view) {
        // Загрузка значения по ключу SAVED_TEXT
        String text = preferences.getString(SAVED_TEXT, "Empty");
        textView.setText(text);
    }
}
```

Запустите приложение и сохраните значение из TextView. Затем удалите приложение из памяти и запустите новый экземпляр приложения. Нажмите кнопку «loadText».

Для просмотра сохранённых данных возможно открыть файловый менеджер устройства в android studio: View | Tool Windows | Device File Explorer и перейти в папку data/data/package_name/shared_prefs. Открыть файл MainActivity.xml.

Стоит обратить внимание, что сами данные хранятся в нешифрованном виде, поэтому, если на устройстве присутствуют права суперпользователя, то другие приложения смогут получать доступ к этим файлам. На текущий момент компания Google разрабатывает новый механизм EncryptedSharedPreferences, поддерживающий шифрование (этап альфа тестирования) .

3 РАБОТА С ФАЙЛАМИ.

Создать новый модуль. Название internalFileStorage.

Работа с preferences позволяет сохранить небольшие данные отдельных типов

(string, int), но для работы с большими массивами данных, такими как графически файлы, файлы мультимедиа и т.д., производится обращение к файловой системе.

В пути к файлам в качестве разграничителя в Linux использует прямой слеш «/», а не обратный «\» (как в Windows). Все названия файлов и каталогов являются регистрозависимыми, то есть «data» не является «Data».

Приложение Android сохраняет свои данные в каталоге /data/data/<название_пакета>/ и, как правило, относительно этого каталога будет идти работа. По умолчанию все файлы, которые там располагаются, доступны только тому приложению, которое их создало. Разумеется, возможно сделать файлы доступными для других приложений, но для этого требуются специальные действия. Если приложение не открывает файлы для доступа извне, получить доступ к ним возможно будет только получив root.

Для работы с файлами абстрактный класс android.content.Context определяет ряд методов:

- deleteFile(String name): удаление определенного файла;
- fileList(): получение всех файлов, которые содержатся в подкаталоге */files* в каталоге приложения;
- getCacheDir(): получение ссылки на подкаталог *cache* в каталоге приложения;
- getDir(String dirName, int mode): получение ссылки на подкаталог в каталоге приложения, если такого подкаталога нет, то он создается;
- getExternalCacheDir(): получение ссылки на папку */cache* внешней файловой системы устройства;
- getExternalFilesDir(): получение ссылки на каталог */files* внешней файловой системы устройства;
- getFileStreamPath(String filename): возвращение абсолютного пути к файлу в файловой системе;
- openFileInput(String filename): открытие файла для чтения;
- openFileOutput (String name, int mode): открытие файла для записи.

Все файлы, которые создаются и редактируются в приложении, как правило,

хранятся в подкаталоге /files в каталоге приложения.

Для непосредственного чтения и записи файлов применяются также стандартные классы Java из пакета java.io.

Система позволяет создавать файлы с двумя разными режимами:

- MODE_PRIVATE – файлы могут быть 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.

```
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();
        }
    }
}
```

Для просмотра файла на эмуляторе требуется открыть следующий экран: View->Tool Windows->Device File Explorer.

В папке data->data требуется найти папку проекта и открыть файл:

Для чтения файла применяется поток ввода InputStream. Базовый класс InputStream представляет классы, которые получают данные из различных источников:

- массив байтов;
- строка (String);
- файл;
- канал (pipe): данные помещаются с одного конца и извлекаются с другого;
- последовательность различных потоков, которые можно 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 – позволяет использовать буфер в памяти (массив байтов) в качестве источника данных для входного потока;
- DataInputStream – входной поток, включающий методы для чтения стандартных типов данных Java;
- FileInputStream – для чтения информации из файла;
- FilterInputStream – абстрактный класс, предоставляющий интерфейс для классов-надстроек, которые добавляют к существующим потокам полезные свойства;
- InputStream – абстрактный класс, описывающий поток ввода;
- ObjectInputStream – входной поток для объектов;
- StringBufferInputStream – превращает строку (String) 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.

Методы класса:

– int available() - возвращает количество байтов ввода, доступные в данный момент для чтения;

– close() - закрывает источник ввода. Следующие попытки чтения передадут исключение IOException;

– void mark(int readlimit) - помещает метку в текущую точку входного потока, которая остаётся корректной до тех пор, пока не будет прочитано readlimint байт;

– boolean markSupported() - возвращает *true*, если методы mark() и reset() поддерживаются потоком;

– int read() - возвращает целочисленное представление следующего доступного байта в потоке. При достижении конца файла возвращается значение -1

– int read(byte[] buffer) - пытается читать байты в буфер, возвращая количество прочитанных байтов. По достижении конца файла возвращает значение -1;

– int read(byte[] buffer, int byteOffset, int byteCount) - пытается читать до *byteCount* байт в *buffer*, начиная с смещения *byteOffset*. По достижении конца файла возвращает -1;

– reset() - сбрасывает входной указатель в ранее установленную метку;

– long skip(long byteCount) - пропускает *byteCount* байт ввода, возвращая количество проигнорированных байтов.

В метод onCreate требуется добавить вызов метода getTextFromFile, который считывает файл и возвращает строку. В данном методе используется многопоточность, изученная в 4 практическом занятии.

```
@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;
  }
}
```

После открытия файла содержимые данные отобразятся через 5 секунд в TextView.

3.1 Внешнее хранилище

Поскольку внешнее хранилище может быть недоступно – например при подключении устройства к компьютеру или при удалении SD карты, требуется всегда проверять раздел на доступность, прежде чем его использовать. Имеется возможность запросить состояние внешнего хранилища с помощью метода getExternalStorageState(). Если 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 (Практика №5):

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

Для хранения данных в публичном каталоге используется следующая директория (в данном случае в общем каталоге *pictures*):

```
public File getAlbumStorageDir(String albumName) {
    // Получение публичного каталога картинок
    File file = new File(Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

3.2 Задание

Создать новый модуль: Notebook

Требуется создать приложение – «Блокнот» только с сохранением файлов. В одном поле пользователь вводит «ИмяФайла», в другом текст. При следующем открытии приложения данные загружаются из последнего файла в EditText. Последний путь до папки требуется хранить в SharedPreference.

4 БАЗА ДАННЫХ SQLITE

4.1 SQLite

SQLite — легковесный фреймворк, который, с одной стороны позволяет по максимуму использовать возможности SQL, с другой — бережно относится к ресурсам устройства. Проект с открытыми исходными кодами, поддерживающий стандартные возможности обычной SQL: синтаксис, транзакции и др. Занимает мало места - около 250 кб. Домашняя страница SQLite: http://www.sqlite.org.

SQLite поддерживает типы TEXT (аналог String в Java), INTEGER (аналог long в Java) и REAL (аналог double в Java). Остальные типы следует конвертировать, прежде чем сохранять в базе данных. SQLite сама по себе не проверяет типы данных, поэтому разработчик может записать целое число в колонку, предназначенную для строк, и наоборот.

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

Отсутствует тип данных, работающий с датами. Возможно использование строковых значений, например, как 2018-12-10 (10 декабря 2018 года). Для даты со временем рекомендуется использовать формат 2018-12-10T09:10. В таких случаях возможно использование некоторых функций SQLite для добавления дней, установки начала месяца и т.д. SQLite не поддерживает часовые пояса. Также не поддерживается тип boolean. Возможно использование числа 0 для false и 1 для true.

Не рекомендуется для использования тип BLOB для хранения данных (картинки) в Android. Лучшим решением будет хранить в базе путь к изображениям, а сами изображения хранить в файловой системе.

Для начала работы с базой данных требуется задать необходимые настройки для создания или обновления базы данных.

Так как сама база данных SQLite представляет собой файл, то по сути при

работе с базой данных, разработчик взаимодействует с файлом. Поэтому операции чтения и записи могут быть довольно медленными. Настоятельно рекомендуется использовать асинхронные операции.

Когда приложение создаёт базу данных, она сохраняется в каталоге DATA */data/имя_пакета/databases/имя_базы.db*.

Метод Environment.getDataDirectory() возвращает путь к каталогу DATA.

Основными пакетами для работы с базой данных являются *android.database* и *android.database.sqlite*.

База данных SQLite доступна только приложению, которое создаёт её. Если вы хотите дать доступ к данным другим приложениям, вы можете использовать контент-провайдеры (ContentProvider).

4.2 Классы для работы с SQLite

Работа с базой данных сводится к следующим задачам:

- создание и открытие базы данных;
- создание таблицы;
- создание интерфейса для вставки данных (insert);
- создание интерфейса для выполнения запросов (выборка данных);
- закрытие базы данных.

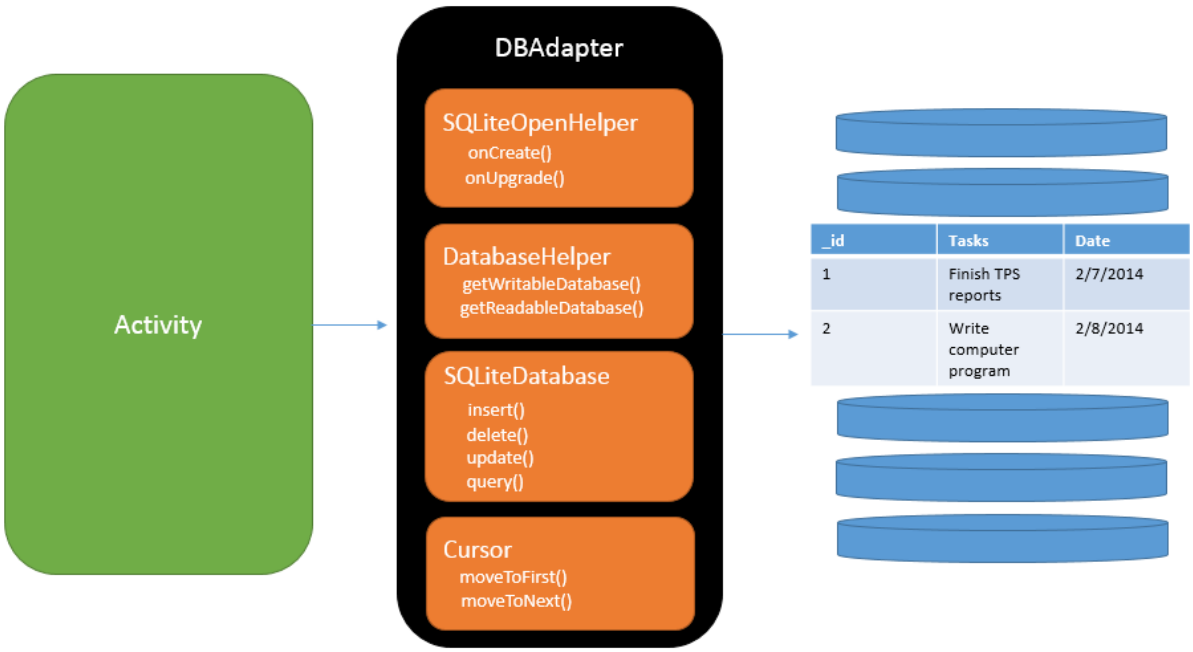


Рисунок 4.1 - Структура механизма взаимодействия приложения с БД

Класс ContentValues

Класс ContentValues используется для добавления новых строк в таблицу. Каждый объект этого класса представляет собой одну строку таблицы и выглядит как ассоциативный массив с именами столбцов и значениями, которые им соответствуют.

Курсоры

В Android запросы к базе данных возвращают объекты класса Cursor. Вместо того чтобы извлекать данные и возвращать копию значений, курсоры ссылаются на результирующий набор исходных данных. Курсоры позволяют управлять текущей позицией (строкой) в результирующем наборе данных, возвращаемом при запросе.

Класс SQLiteOpenHelper: Создание базы данных

Библиотека Android содержит абстрактный класс SQLiteOpenHelper, с помощью которого можно создавать, открывать и обновлять базы данных. Это основной класс, с которым вам придётся работать в своих проектах. При реализации этого вспомогательного класса от вас 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)
- onOpen(SQLiteDatabase)
- getReadableDatabase()
- getWritableDatabase()

SQLiteDatabase

Для управления базой данных SQLite существует 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, если вы хотите провести эти (или любые другие) операции вручную.

Каждый раз, когда вы редактируете очередное значение из базы данных, нужно вызывать метод 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()* передают семь параметров. Если какой-то параметр для запроса вас не интересует, то оставляете *null*:

- table — имя таблицы, к которой передается запрос;
- String[] columnNames — список имен возвращаемых полей (массив). При передаче *null* возвращаются все столбцы;
- String whereClause — параметр, формирующий выражение WHERE (исключая сам оператор WHERE). Значение null возвращает все строки. Например: *_id = 19 and summary = ?;*
- String[] selectionArgs — значения аргументов фильтра. Вы можете включить ? в "whereClause"". Подставляется в запрос из заданного массива;
- String[] groupBy - фильтр для группировки, формирующий выражение GROUP BY (исключая сам оператор GROUP BY). Если GROUP BY не нужен, передается null;
- String[] having — фильтр для группировки, формирующий выражение HAVING (исключая сам оператор HAVING). Если не нужен, передается null;
- String[] orderBy — параметр, формирующий выражение ORDER BY (исключая сам оператор ORDER BY). При сортировке по умолчанию передается null.

Реализация взаимодействия с БД SQLite будет рассмотрена далее с использованием компонента Room.

4.3 Data Access Object (DAO)

Шаблон Data Access Object (DAO) является структурным шаблоном, который позволяет изолировать прикладной/бизнес-уровень 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 том, чтобы скрыть от приложения все сложности, связанные с выполнением операций CRUD в базовом механизме хранения. Это позволяет обоим слоям развиваться отдельно, ничего не зная друг о друге.

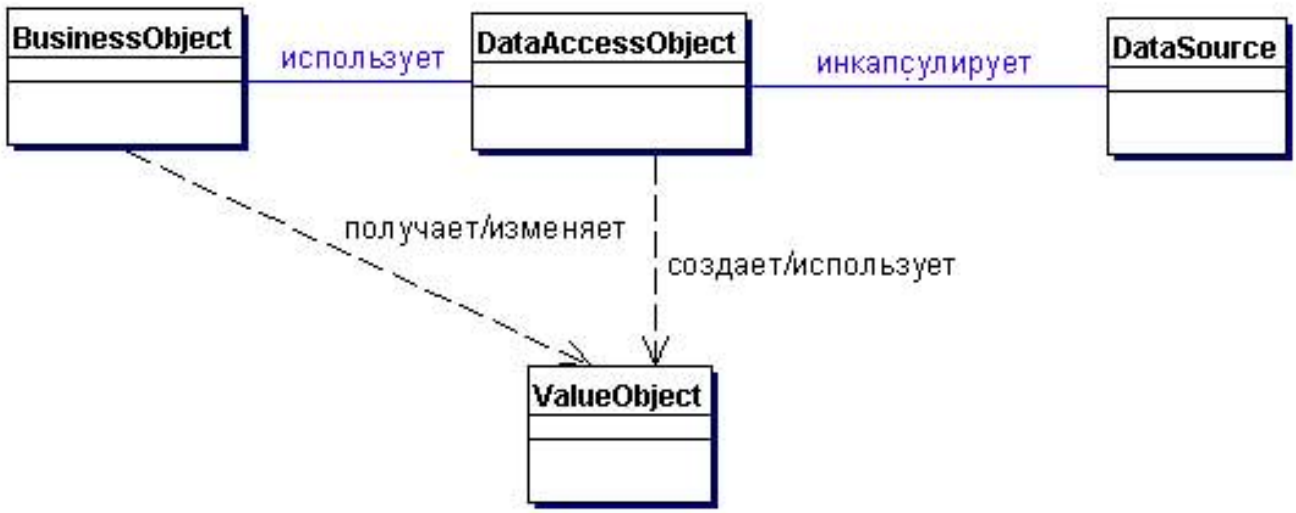


Рисунок 4.2 - Архитектура паттерна DAO

На рисунке 4.2 представлена архитектура паттерна DAO. Данный паттерн состоит из:

– BusinessObject представляет клиента данных. Это объект, который нуждается в доступе к источнику данных для получения и сохранения данных;

– DataAccessObject является первичным объектом данного паттерна. DataAccessObject абстрагирует используемую реализацию доступа к данным для BusinessObject, обеспечивая прозрачный доступ к источнику данных. BusinessObject передает также ответственность за выполнение операций загрузки и сохранения

данных объекту DataAccessObject;

– DataSource представляет реализацию источника данных. Источником данных может быть база данных, например, СУБД, XML документы, данные в формате JSON и др;

– ValueObject представляет собой Transfer Object, используемый для передачи данных. DataAccessObject может использовать Transfer Object для возврата данных клиенту. DataAccessObject может также принимать данные от клиента в объекте Transfer Object для их обновления в источнике данных.

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

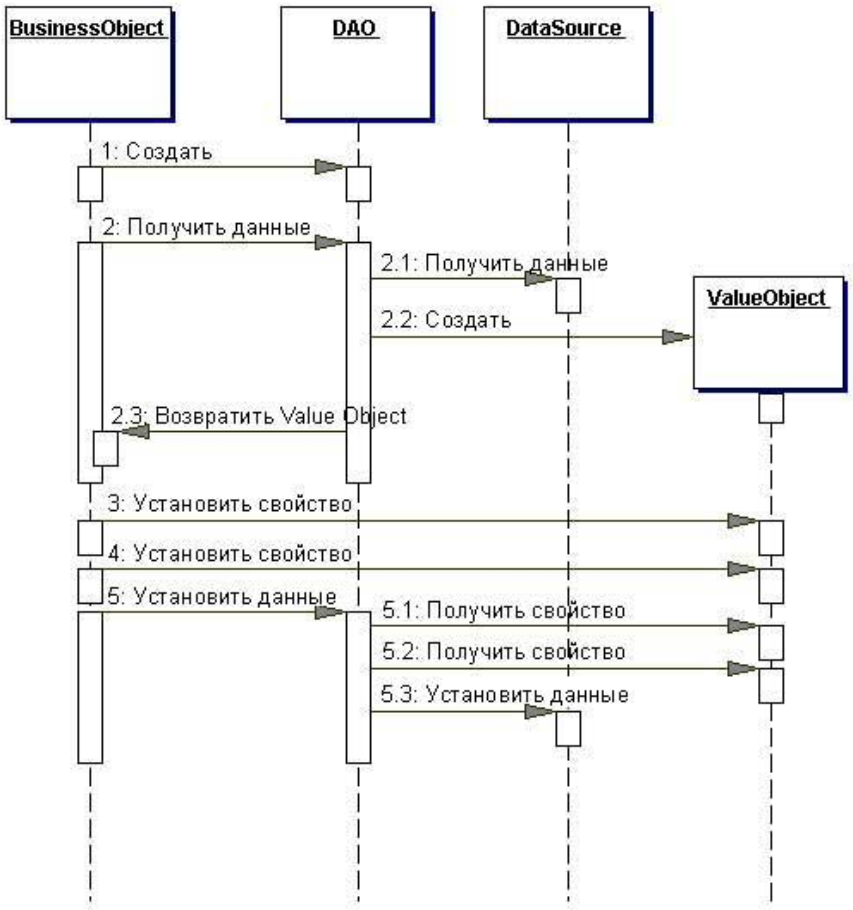


Рисунок 4.3 - Диаграмма последовательности действий паттерна Data Access Object

4.4 Room

Работа с базой данных SQLite в Android не отличается удобством, требуется большое количество строчек кода. Сторонние разработчики предлагали собственные варианты. Наиболее популярными стали такие библиотеки как Realm,

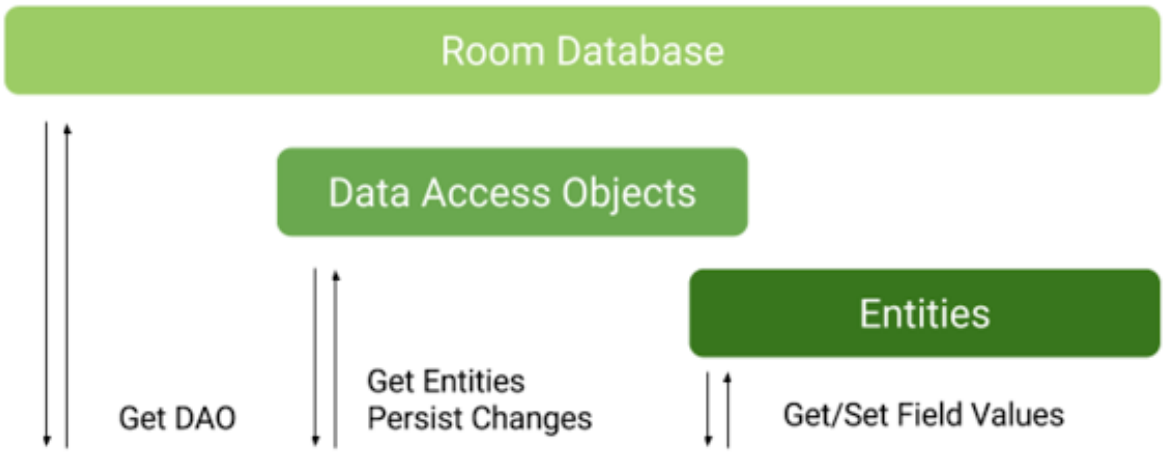
ORMLite, GreenDao, DBFlow, Sugar ORM.

Компонент Room Persistence — предоставляет удобную обертку для работы с базой данных SQLite. Room входит в состав Android Architecture Components и является обёрткой над SQLiteOpenHelper, т.е. ORM (Object-relational mapping) между классами Java и SQLite. Минимальная версия для работы с Room - API 15.

Компонент возможно разделить на три части: Entity, DAO (Data Access Object), Database.

Entity — объектное представление таблицы. С помощью аннотаций можно легко и без лишнего кода описать поля.

Для создания Entity требуется создать класс POJO (Plain Old Java Object). Пометить класс аннотацией Entity.



4.5 Задание

Создать модуль. Название приложения Room. Создать базу данных для хранения информации о сотрудниках компании.

В build.gradle файле модуля требуется добавить dependencies и синхронизировать проект:

```
def room_version = "2.2.5"

implementation "androidx.room:room-runtime:$room_version"
annotationProcessor "androidx.room:room-compiler:$room_version"
```

ШАГ 1 - создание класса, на основании которого будет создана таблица базы данных. Room использует аннотации, на основе которых генерируется таблица базы данных. Аннотация *@Entity* создаёт таблицу с именем, совпадающим с именем

класса. У *@Entity* обязательно должно быть поле с первичным ключом. Указывается через аннотацию *@PrimaryKey*. Возможно добавить дополнительный атрибут для генерации уникального идентификатора autoGenerate (аналог ключевого слова AUTOINCREMENT в SQL), либо разработчик должен следить за уникальностью. Поля таблицы создаются в соответствии с полями класса.

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

ШАГ 2 - в созданном классе есть вся информация, которая требуется Room для создания таблиц. Требуется механизм для управления данными (добавлять, удалять, делать запросы). Для данной цели служит интерфейс DAO (Data Access Object). Создаётся новый класс-интерфейс.

```
@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 позволяют получить полный список сотрудников или конкретного сотрудника по id. Обратите внимание, что в качестве имени таблицы используется employee. Имя таблицы равно имени Entity класса, т.е. Employee, но в SQLite не важен регистр в именах таблиц, поэтому возможно писать employee. Аннотация *@Insert* вставляет данные, есть ещё вариант @Insert(onConflict = REPLACE – при обнаружении записи с одинаковыми уникальными индексами заменяет содержимое). Аннотации *@Update* и *@Delete* обновляют и удаляют записи. Аннотация *@Query* позволяет писать запросы на языке SQL. Запросы в аннотациях проверяются на этапе компиляции, если разработчик допустит ошибку или опечатку, то среда разработки заметит и предложит исправить до запуска приложения.

ШАГ 3 - необходимо создать абстрактный класс базы данных, наследующий от RoomDatabase, который будет отвечать за ведение самой базы данных и предоставление экземпляров DAO. Требуется указать класс-модель и номер версии базы данных. Класс должен содержать абстрактный метод без параметров и возвращать класс, аннотированный как *@Dao* из предыдущего шага.

Требуется менять номер версии при любой модификации модели данных.

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

ШАГ 4 - создать класс App наследуемый от класса Application. Данный класс является реализацией паттерна Singleton. Класс App создаётся один раз при запуске приложения и предназначен для инициализации компонентов уровня приложения и хранения состояния. Требуется вызвать контекстное меню у папки, в которой размещён код приложения: File|New|Java/Kotlin Class. Присвоить имя App. Далее требуется перейти в manifest – файл и установить следующее значение:

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

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

Добавить в App следующий код:

```
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 в методе

onCreate. Необходимыми аргументами для создания объекта базы данных является ApplicationContext, а также указывается AppDatabase класс и имя файла для базы. Учитывайте, что при вызове данного кода Room каждый раз будет создавать новый экземпляр AppDatabase. Эти экземпляры достаточно массивные и рекомендуется использовать один экземпляр для всех операций. Поэтому необходимо реализовать доступ к данному объекту. Работать с базой данных следует в отдельном потоке. Имеется возможность выполнения запросов в главном потоке для не ресурсоёмких задач – метода allowMainThreadQueries(), однако это является нерекомендуемым способом работы с БД. В качестве инструмента обмена данными между слоями, как правило, используется RxJava или компонент из Architecture Components - LiveData.

ШАГ 5 – работа с базой.

```
public class MainActivity extends AppCompatActivity {
    private String TAG = MainActivity.class.getSimpleName();
    @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 «Настройки», в котором пользователь должен указать определённые параметры (задумка исполнителя) и сохранить их в SharedPreferences. Добавить фрагмент «Истории». Данный экран должен отобразить RecycleView с историями. При нажатии на Floating Action Button вызывается диалоговое окно/фрагмент/View

создания истории с кнопкой сохранения.