Лабораторная работа № 4
Работа с хранилищами данных.

1. Язык Kotlin.

1.1. **Классы данных**

Существует категория классов, настолько распространенных, что в Java для них есть имя: они называются POJO (Plain Old Java Object), или старые добрые Java-объекты. Это простые представления структурированных данных. Они представляют собой набор членов данных (полей), у большинства из которых есть методы чтения и записи, а также несколько других методов: equals, hashCode и toString. Такого рода классы настолько распространены, что Kotlin сделал их частью языка. Они называются классами данных. Можно улучшить определение класса Point, сделав его классом данных:

```
data class Point(var x: Int, var y: Int? =3)
```

1.2 **Классы-одиночки (Синглтоны)**

Часто бывает необходимо создать единственный экземпляр данного класса. Такой экземпляр называется синглтоном (одиночкой). Шаблон проектирования «Синглтон» – это метод, гарантирующий невозможность создания более одного экземпляра класса. В Java этот шаблон имеет сложную реализацию, потому что трудно гарантировать создание не более одного экземпляра. В Kotlin синглтон легко создать, заменив слово class словом object:

```
object MyWindowAdapter: WindowAdapter() {
   override fun windowClosed(e: WindowEvent?) {
   TODO("не реализовано")
}}
```

Объект object не имеет конструкторов. Если в объекте есть свойства, они должны быть инициализированы или объявлены абстрактными.

1.3. **Объекты-компаньоны**

Классы в Kotlin не имеют статических членов. Чтобы сымитировать их, нужно создать специальную конструкцию, называемую объектом-компаньоном, или сопутствующим объектом:

```
data class Person(val name: String, val registered: Instant = Instant.now()) {
  companion object {
      fun create(xml: String): Person {
         TODO("TODO")
      }
   }
}
```

Функцию create можно вызвать относительно вмещающего класса, подобно статическим методам в Java:

```
Person.create(someXmlString)
```

Ее также можно вызвать относительно объекта-компаньона, но в этом нет необходимости:

```
Person.Companion.create(someXmlString)
```

Чтобы вызвать эту функцию из кода на Java, обязательно нужно сослаться

на объект-компаньон. Чтобы получить возможность вызывать его как статический метод класса, достаточно добавить аннотацию @JvmStatic.

Объект-компаньон— это объект-одиночка, всегда связанный с классом Kotlin. Хотя это и не обязательно, чаще всего определение объекта-компаньона размещается в нижней части связанного с ним класса.

Объекты-компаньоны могут иметь имена, расширять классы и наследовать от интерфейсов. В данном примере объект-компаньон TimeExtension называется StdTimeExtension и наследует от интерфейса Formatter:

```
interface Formatter {
   val yearMonthDate: String
}
class TimeExtensions {
   // другой код
   companion object StdTimeExtension : Formatter {
      const val TAG = "TIME_EXTENSIONS"
      override val yearMonthDate = "yyyy-MM-d"
   }
}
```

2. Работа с хранилищами данных в Android.

2.1 SharedPreferences

SharedPreferences - механизм постоянного хранения данных приложения, который позволяет сохранять и восстанавливать данные и настройки приложения и activity на стадиях жизненного цикла и между запусками приложения.

Данные хранятся в виде пар «ключ - значение».

Физически SharedPreferences реализовано в виде файла в котором данные хранятся в виде пар «ключ - значение». Данный файл доступен только конкретному приложению, которым он создавался.

Для доступа к хранилищу используется класс, экземпляр которого можно получить через методы:

getPreferences(int mode**)** - вызывается внутри activity и содержит данные, относящиеся к ней;

getSharedPreferences(String name, int mode**)** - содержит определенные для всего приложения данные, вызывается из любого контекста.

Для записи данных в SharedPreferences используется объект класса **SharedPreferences**.**Edit**. Его возвращает метод **edit**() объекта SharedPreferences.

В зависимости от типа сохраняемых данных используются методы:

putBoolean(String key, boolean value**)**;

putFloat(String key, float value**);**

putInt(String key, int value**);**

putLong(String key, long value**);**

putString(String key, String value**);**

putStringSet(String key, Set values**).**

После добавления данных необходимо вызвать метод **apply()** или **commit()**. Метод **apply()** изменяет данные в памяти и асинхронно записывает обновления в файл. Метод **commit()** осуществляет синхронную запись данных в файл, его вызова из основного потока следует избегать при записи большого объема данных, т.к. это может замедлить работу пользовательского интерфейса.

Для получения данных из SharedPreferences используются методы:

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**)**;
getStringSet(String key, Set<String> defValues**)**;
getAll().

В указанные методы передается значение ключа и, при необходимости, возвращаемое значение по умолчанию, если ключ отсутствует.

Например, если нам необходимо сохранить текст из текстового поля и затем его загрузить, можно использовать следующий код:

```
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   setContentView(R.layout.activity_helloact)

   val text1: TextView = findViewById(R.id.text1)
   val button1: Button = findViewById(R.id.button1)

   val sharedPref: SharedPreferences = getPreferences(Context.MOD-
E_PRIVATE)
   val editor: SharedPreferences.Editor = sharedPref.edit()
   editor.putString("text", "Shared ${text1.text}").apply()

   button1.setOnClickListener {
      text1.text = sharedPref.getString("text", "")
   }
}
```

2.2 SQLite для хранения данных

SQLite - реляционная СУБД, которая входит в состав стандартной библиотеки ОС Android и используемая для хранения данных в приложениях.

Библиотека android.database.sqlite предоставляет низкоуровневый API для работы с реляционной БД.

Для работы с БД Google рекомендует реализовать класс контракта,

который описывает организацию базы данных.

Например, для БД, хранящей данные пользователей, такой класс может выглядеть следующим образом:

```
import android.provider.BaseColumns

object DBContract {
    object UserEntry : BaseColumns {
        const val TABLE_NAME = "users"
        const val COLUMN_NAME_KEY_ID = "id"
        const val COLUMN_NAME_LOGIN = "login"
        const val COLUMN_NAME_PASS = "pass"
    }
}
```

class UserEntry в данном контракте содержит константы, задающие имена таблицы для хранения данных пользователей и названия ее столбцов. Эти константы будут использоваться при построении SQL-запросов.
Наличие контракта позволяет иметь одну точку редактирования описания схемы данных, т.е. использовать одни и те же константы во всех классах приложения.

Для доступа к БД используется класс SQLiteOpenHelper, который содержит методы для управления базой данных. Экземпляр БД можно получить, вызвав метод getWritableDatabase() или getReadableDatabase() у класса-наследника SQLiteOpenHelper.
Таким образом, в приложении необходимо создать подкласс SQLiteOpenHelper, который переопределяет методы onCreate() и onUpgrade(). Также можно реализовать методы onDowngrade() или onOpen().

```
class DatabaseHandler(context: Context) :
    SQLiteOpenHelper(context, DATABASE_NAME, null,
DATABASE_VERSION) {

    companion object {
        private const val DATABASE_VERSION = 1
        private const val DATABASE_NAME = "Users.db"
    }

    override fun onCreate(db: SQLiteDatabase) {
        val createUsersTable = "CREATE TABLE ${DBContract.UserEn-
try.TABLE_NAME} (" +
```

```
            "${DBContract.UserEntry.COLUMN_NAME_KEY_ID} INTEGER
PRIMARY KEY," +
            "${DBContract.UserEntry.COLUMN_NAME_LOGIN} TEXT," +
            "${DBContract.UserEntry.COLUMN_NAME_PASS} TEXT)"
        db.execSQL(createUsersTable)
    }

    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int,
newVersion: Int) {
        db.execSQL("DROP TABLE IF EXISTS ${DBContract.UserEn-
try.TABLE_NAME}")
        onCreate(db)
    }

    fun addUser(user: User) {
        val db = this.writableDatabase
        val values = ContentValues()
        values.put(DBContract.UserEntry.COLUMN_NAME_LOGIN, user.lo-
gin)
        values.put(DBContract.UserEntry.COLUMN_NAME_PASS, user.-
pass)
        db.insert(DBContract.UserEntry.TABLE_NAME, null, values)
    }

    fun deleteAll() {
        writableDatabase.execSQL("DELETE FROM ${DBContract.UserEn-
try.TABLE_NAME}")
    }

}
```

Помимо переопределенных методов onCreate() и onUpgrade() созданный класс реализует методы добавления и выборки данных из БД, например, метод добавления нового пользователя addUser.

Реализация данного метода использует getWritableDatabase для получения экземпляра класса, работающего с БД. Затем создается класс ContentValues, в котором формируются данные в форме ключ-значение, пригодной для добавления в БД. Например, метод put используется для добавления данных в столбец таблицы. При этом ключом является название столбца таблицы, а

значение – это добавляемые данные.

Метод insert осуществляет непосредственное добавление данных в таблицу.

Аналогично реализуются методы выборки данных, в которых используется метод query. Этот метод возвращает данные в виде объекта Cursor, для навигации по которому используются методы:

getCount() - получает количество извлеченных из базы данных объектов;

moveToFirst() и **moveToNext**() позволяют переходить к первому и к следующему элементам выборки;

isAfterLast() позволяет проверить, достигнут ли конец выборки.

Также имеется набор методов для чтения каждого типа данных: **getString**(), **getInt**() и **getFloat**().

Например, так будет выглядеть метод загрузки из БД всех строк таблицы Users:

```
fun getAllUsers(): List<User> {
    val userList = mutableListOf<User>()
    val selectQuery = "SELECT * FROM ${DBContract.UserEntry.TABLE_-
NAME}"
    val db = this.writableDatabase
    val cursor: Cursor = db.rawQuery(selectQuery, null)
    if (cursor.moveToFirst()) {
        do {
            val id = cursor.getString(
                cursor.getColumnIndexOrThrow(DBContract.UserEntry.-
COLUMN_NAME_KEY_ID)
            ).toInt()
            val login = cursor.getString(
                cursor.getColumnIndexOrThrow(DBContract.UserEntry.-
COLUMN_NAME_LOGIN)
            )
            val pass = cursor.getString(
                cursor.getColumnIndexOrThrow(DBContract.UserEntry.-
COLUMN_NAME_PASS)
            )
            val user = User(id = id, login = login, pass = pass)
            userList.add(user)
        } while (cursor.moveToNext())
    }
    cursor.close()
    return userList
```

```
}
```

Данные каждой строки загружаются в отдельный экземпляр класса User, который описывает модель (бизнес-логику) данных.

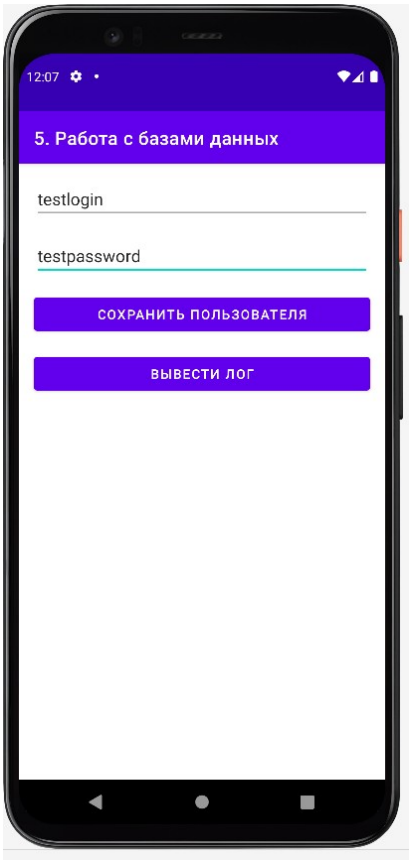
```
data class User(
    val id: Int = -1,
    val login: String,
    val pass: String
)
```

Для работы с БД из класса активити (fragment или сервиса) необходимо создать экземпляр класса-наследника SQLiteOpenHelper, после чего можно использовать его методы (например, addUser или getAllUsers) для работы с БД:

```
class HelloActivity : AppCompatActivity() {

    companion object {
        private val TAG: String = HelloActivity::class.java.simpleName
    }

    private lateinit var etLogin: EditText
    private lateinit var etPass: EditText
    private lateinit var btnSave: Button
    private lateinit var btnLoad: Button

    private val db = DatabaseHandler(this)

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_helloact)

        title = "Работа с базами данных"

        etLogin = findViewById(R.id.et_login)
        etPass = findViewById(R.id.et_pass)
        btnSave = findViewById(R.id.btn_save)
        btnLoad = findViewById(R.id.btn_load)

        btnSave.setOnClickListener {
            val login = etLogin.text.toString()
            val pass = etPass.text.toString()
```

```
            val user = User(login = login, pass = pass)
            db.addUser(user)
        }

        btnLoad.setOnClickListener {
            val usersLog = db.getAllUsers().joinToString(separator = ",\n")
            Log.d(TAG, "Users:\n $usersLog")
        }

    override fun onPause() {
        super.onPause()
        db.deleteAll()
    }

    override fun onDestroy() {
        super.onDestroy()
        db.close()
    }
}
```

На экране есть два поля для ввода данных – логин и пароль пользователя. Кнопка btnSave отвечает за сохранение введенных данных и добавления записи в бд. Кнопка btnLoad выводит лог с информацией о всех пользователях в таблице. При сворачивании приложения все данные удаляются из бд. Подключение к базе данных рекомендуется оставлять открытым до тех пор, пока может потребоваться доступ к ней. Обычно оптимально закрывать базу данных в onDestroy() в activity.



Модель данных User представлена в качестве учебного примера, в реальных приложениях не стоит хранить пароль пользователя в открытом виде в базе данных

2.3 Работа со списками

Во многих приложениях используются списки. Для работы со списками можно использовать ListView, однако в нем есть большое количество недостатков и в качестве более современного и гибкого решения Google рекомендует использовать RecyclerView. Для создания списка необходимо выполнить следующие шаги:

1. Добавить RecyclerView в файл с верстkой
2. Создать верстку для одного элемента списка
3. Создать адаптер для списка
4. Установить адаптер для добавленного RecyclerView

В качестве примера, создадим список для отображения добавленных в базу данных пользователей.

1. Добавим RecyclerView в верстку для activity

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/activity_hello"
```

```
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <... />

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/rv_users"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        android:layout_marginTop="16dp"
        app:layout_constraintTop_toBottomOf="@+id/btn_load" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

2. В папке layout создадим верстку для одного элемента списка – item_user.xml. Будем отображать id, логин и пароль пользователя. Также отрисуем иконку, при нажатии на которую будет показываться меню с дополнительными действиями для элемента.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <TextView
        android:id="@+id/tv_id"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:textColor="@android:color/black"
        android:textSize="50sp"
        android:textStyle="bold"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
```

```
        app:layout_constraintTop_toTopOf="parent"
        tools:text="0" />

    <TextView
        android:id="@+id/tv_login"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:textStyle="bold"
        app:layout_constraintStart_toEndOf="@+id/tv_id"
        app:layout_constraintTop_toTopOf="@+id/tv_id"
        tools:text="Login" />

    <TextView
        android:id="@+id/tv_pass"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        app:layout_constraintStart_toEndOf="@+id/tv_id"
        app:layout_constraintTop_toBottomOf="@+id/tv_login"
        tools:text="Password" />

    <ImageButton
        android:id="@+id/ib_more"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:background="@android:color/transparent"
        android:src="@drawable/ic_baseline_more_vert_24"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

3. Создаем адаптер для списка. Класс адаптера должен наследоваться от RecyclerView.Adapter<T>, где T – класс, являющийся наследником RecyclerView.ViewHolder и представляющий собой один элемент списка. В этом классе будет сконцентрировано связывание данных с view-элементами и

добавление слушателей.

```
class UserAdapter : RecyclerView.Adapter<User-
Adapter.UserViewHolder>() {

    private var users = listOf<User>() // список пользователей

    override fun onCreateViewHolder(parent: ViewGroup, viewType:
Int): UserViewHolder {
        val view = LayoutInflater.from(parent.context).inflate(R.lay-
out.item_user, parent, false)
        return UserViewHolder(view)
    }

    override fun onBindViewHolder(holder: UserViewHolder, posi-
tion: Int) = holder.bind(users[position])

    override fun getItemCount(): Int = users.size

    inner class UserViewHolder(itemView: View) : Recy-
clerView.ViewHolder(itemView) {

        fun bind(user: User) {


        }
}
```

После создания адаптера, необходимо переопределить 3 метода:

- **onCreateViewHolder** – метод должен вернуть объект класса наследника ViewHolder – UserViewHolder. Тут же происходит парсинг файла item_user.xml.
- **onBindViewHolder** – в данном методе происходит связывания данных с view-элементами. Так как связывание мы решили сконцентрировать в классе UserViewHolder, то просто вызовем для объекта функцию bind и передадим туда объект класса User.
- **getItemCount** – метод должен возвращать количество элементов списка

В функции bind класса UserViewHolder будет происходить связывание данных с view-элементами. Реализация будет выглядеть следующим образом:

```
fun bind(user: User) {
    itemView.findViewById<TextView>(R.id.tv_id).text =
```

```
user.id.toString()
    itemView.findViewById<TextView>(R.id.tv_login).text = user.login
    itemView.findViewById<TextView>(R.id.tv_pass).text = user.pass
}
```

В классе адаптера UserAdapter добавим функцию setData, которая будет обновлять данные в списке

```
class UserAdapter : RecyclerView.Adapter<User-
Adapter.UserViewHolder>() {

    ...

    fun setData(userList: List<User>) {
        users = userList
        notifyDataSetChanged()
    }
}
```

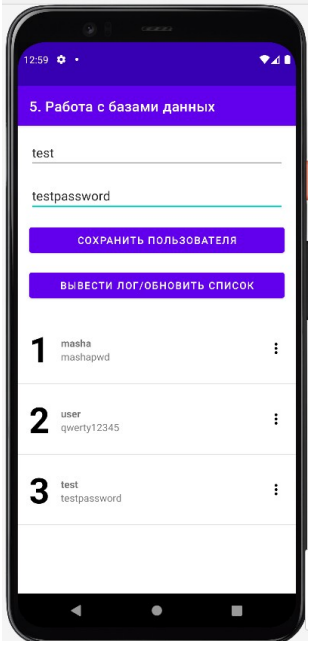
В качестве примера для обновления данных используется функция **notifyDataSetChanged**, однако она обладает недостатком – при вызове происходит обновление всех элементов в списке, даже если данные для элемента не поменялись. Для большей производительности можно использовать класс DiffUtil для вычисления «разности» между двумя списками или использовать встроенные методы для обновления элементов.
notifyItemChanged, **notifyItemInserted**, **notifyItemRemoved** – методы для изменения, вставки и удаления элемента (обновляется один элемент, а не целый список). С полным списком всех функций можно ознакомиться в документации.

4. Настроим список в классе нашей активности:

```
class HelloActivity : AppCompatActivity() {

    private lateinit var rvUsers: RecyclerView
    private val userAdapter = UserAdapter()

    override fun onCreate(savedInstanceState: Bundle?) {
        ...

            btnLoad.setOnClickListener {
                val users = db.getAllUsers()
                val usersLog = users.joinToString(separator = ",\n")
```

```
            Log.d(TAG, "Users:\n $usersLog")
            userAdapter.setData(users)
        }


        rvUsers = findViewById(R.id.rv_users)
        rvUsers.layoutManager = LinearLayoutManager(this)
        rvUsers.addItemDecoration(DividerItemDecoration(this, Di-
viderItemDecoration.VERTICAL))
        rvUsers.adapter = userAdapter
    }
}
```

Устанавливаем для RecyclerView режим отображения – в нашем случае список будет отображаться линейно, поэтому используем LinearLayoutManager.
addItemDecoration – необязательная функция, которая используется для добавления разделителя между элементами списка. Используем стандартный разделитель DividerItemDecoration.
В конце устанавливаем созданный адаптер в RecyclerView.
Обновление списка будет происходить при нажатии на кнопку btnLoad.



2.4 Взаимодействие со списком

Обработку события нажатия на элемент списка можно реализовать несколькими способами. Классический способ – использовать интерфейс. Определим интерфейс UserClickListener с двумя функциями – для нажатия на элемент и для нажатия на пункт из меню с дополнительными действиями:

```
interface UserClickListener {
```

```
    fun onItemClick(user: User)
    fun onMenuDeleteClick(user: User)
}
```

В классе адаптера UserAdapter определим слушатель и создадим метод setListener для его установки:

```
private var listener: UserClickListener? = null

fun setListener(listener: UserClickListener) {
    this.listener = listener
}
```

В классе UserViewHolder обработаем события нажатия на кнопки:

```
fun bind(user: User) {
    ...


itemView.findViewById<ImageButton>(R.id.ib_more).setOnClickListen
er {
        showPopupMenu(user, it)
    }

    itemView.setOnClickListener {
        listener?.onItemClick(user)
    }}
```

Полная реализация адаптера с обработкой событий выглядит следующим образом:

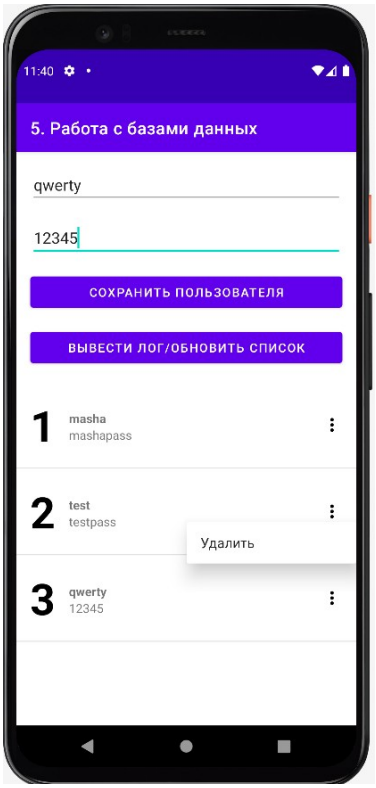
```
class UserAdapter : RecyclerView.Adapter<User-
Adapter.UserViewHolder>() {

    companion object {
        const val MENU_DELETE = 1
    }

    private var listener: UserClickListener? = null
    private var users = listOf<User>()

    fun setListener(listener: UserClickListener) {
```

```
        this.listener = listener
    }

    fun setData(userList: List<User>) {
        users = userList
        notifyDataSetChanged()
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType:
Int): UserViewHolder {
        val view = LayoutInflater.from(parent.context).inflate(R.lay-
out.item_user, parent, false)
        return UserViewHolder(view)
    }

    override fun onBindViewHolder(holder: UserViewHolder, position:
Int) = holder.bind(users[position])

    override fun getItemCount(): Int = users.size

    inner class UserViewHolder(itemView: View) : Recy-
clerView.ViewHolder(itemView) {

        fun bind(user: User) {
            itemView.findViewById<TextView>(R.id.tv_id).text = user.id.-
toString()
            itemView.findViewById<TextView>(R.id.tv_login).text = user.lo-
gin
            itemView.findViewById<TextView>(R.id.tv_pass).text = user.pass

            itemView.findViewById<ImageButton>(R.id.ib_more).se-
tOnClickListener {
                showPopupMenu(user, it)
            }

            itemView.setOnClickListener {
                listener?.onItemClick(user)
            }
```

```
        }

        private fun showPopupMenu(user: User, view: View) {
            val popupMenu = PopupMenu(view.context, view)
            popupMenu.menu.add(0, MENU_DELETE, Menu.NONE,
"Удалить")
            popupMenu.setOnMenuItemClickListener {
                when(it.itemId) {
                    MENU_DELETE -> {
                        listener?.onMenuDeleteClick(user)
                    }
                }
                return@setOnMenuItemClickListener true
            }
            popupMenu.show()
        }
    }
}
```

В классе активности добавим в адаптер события, происходящие при нажатии на элемент и на пункт из меню:

```
userAdapter.setListener(object: UserClickListener {
    override fun onItemClick(user: User) {
        Toast.makeText(this@HelloActivity, "onItemClick() user=$user",
Toast.LENGTH_SHORT).show()
    }

    override fun onMenuDeleteClick(user: User) {
        Toast.makeText(this@HelloActivity, "onMenuDeleteClick()
user=$user", Toast.LENGTH_SHORT).show()
    }
})
```



2.5. Работа с файлами

Приложение Android сохраняет свои данные в каталоге /data/data/<название_пакета>/ и, как правило, относительно этого каталога будет идти работа. Все файлы, которые создаются и редактируются в приложении, как правило, хранятся в подкаталоге /files в каталоге приложения.

Для работы с файлами используется абстрактный класс android.content.Context.

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

```
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.TextView
import java.io.*

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val myTextView = findViewById<TextView>(R.id.textView)
        val myOutput: String
        val myInputStream: InputStream

        try {
            myInputStream = assets.open("MyText.txt")
```

```
            val size: Int = myInputStream.available()
            val buffer = ByteArray(size)
            myInputStream.read(buffer)
            myOutput = String(buffer)

            myTextView.text = myOutput

        } catch (e: IOException) {
            e.printStackTrace()
        }
    }
}
```

3. Задание для самостоятельной реализации

1) Добавить в окно приложения «Меню» кнопку «Администрирование», а в окно «Профиль» пункт «Сделать администратором».

2) Реализовать функционал по регистрации и авторизации для проекта созданного в лабораторных работах №№ 2-3.

Требования:

- при заполнении в окне регистрации полей и нажатии зарегистрироваться, должно происходить регистрация данного пользователя (делается соотвествующая запись в БД);
- первый зарегистрированный пользователь автоматически становиться администратором;
- администратору в окне меню должна отображаться кнопка «Администрирование», а обычным пользователям нет;
- при нажатии на кнопку «Администрирование» должено отображаться окно «Администрирование», представляющее список из лабораторной работы № 2. В списке должны отображаться зарегистрированные пользователи. **При желании** можно улучшить отображение в списке отображая не только ФИО, но и аватар пользователя, а также является ли он администратором.
- При нажатии на любой элемент списка в окне «Администрирование» должно открыться окно «Профиль с данными данного пользователя» и дополнительным пунктом доступным только администратору «Сделать администратором». При активации данного пункта, пользователь тоже становится администратором.
- В окне приветствия зарегистиррованные пользователи после ввода логина и пароля и нажатии кнопки «Вход» должны быть авторизованы как обычный пользователь или администратор.
- Информацию о пользователях хранить в БД.

2) Реализовать функционал по переключению визуальной темы приложения с использованием SharedPreferences. Переключение темы

можно реализовать в окне «Настройки» или в окне «Привествие». Можно комбинировать в окне «Приветствие», для неавторизированных пользователей, а после авторизации взять тему для пользователя оторая была задана в настройках, но в этом случае в SharedPreferences сохранять тему из окна «Приветствие», а из окна «Настройки» хранить в БД для каждого пользователя.

3) НЕОБЯЗАТЕЛЬНОЕ ЗАДАНИЕ:

Реализовать логирование в файл фактов авторизации пользователей с датой и временем. Указаный файл должен читаться и отображаться администратору. Для отображения предусмотреть соответствующие кнопки управления в меню «Администрирования» и окна для отображения.

4. Вопросы для самопроверки

- Для чего используется SharedPreferences?
- Раскажите о принципах работы с SharedPreferences.
- Какие методы есть у SharedPreferences и для чего они используются?
- Как происходит работа с БД в Android приложении?
- Как происходит работа с файлами в Android приложении?
- Для чего используются адаптеры для списков?
- Чем выгодно использовать RecycleView вместо ListView?
- Когда использовать RecycleView, а когда ListView?