Федеральное государственное образовательное бюджетное учреждение высшего образования

«ФИНАНСОВЫЙ УНИВЕРСИТЕТ ПРИ ПРАВИТЕЛЬСТВЕ

РОССИЙСКОЙ ФЕДЕРАЦИИ»

**Колледж информатики и программирования**

Практическая работа

По дисциплине «Разработка мобильных приложений»

Название: «Входной контроль»

Выполнил:

студент группы 3ИСИП-122

Петров Артём Владимирович

Проверил:

Преподаватель Поволяев Д.А.

Москва

2025

СОДЕРЖАНИЕ

[1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 3](#_Toc209690649)

[2. ПРАКТИЧЕСКАЯ ЧАСТЬ 15](#_Toc209690650)

1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Блок А: Основы проекта и жизненный цикл

*– Архитектура проекта:*

1) MainActivity.kt – это основной файл, содержащий логику экрана (Activity). Он написан на языке Kotlin.

1.1) Внутри этого файла код, который управляет поведением пользовательского интерфейса: обработка нажатий кнопок, выполнение вычислений, запросы к сети, взаимодействие с данными. Это "мозг" экрана.

2) AndroidManifest.xml – это главный конфигурационный файл всего приложения. Android-система читает его первым, чтобы узнать основную информацию о приложении.

2.2) Внутри этого файла:

а) объявление всех компонентов приложения (Activity, Service, BroadcastReceiver);

б) разрешения (permissions), которые требуются приложению (например, доступ к интернету или камере);

в) указание главной Activity (точки входа в приложение);

г) минимальная и целевая версия Android SDK.

3) build.gradle – это скрипт для системы сборки Gradle, который управляет зависимостями и настройками сборки конкретно для модуля app

3.1) Внутри этого файла:

а) зависимости (dependencies): список библиотек, которые использует ваш проект (например, androidx.appcompat:appcompat:1.6.1);

б) версии SDK: compileSdk, minSdk, targetSdk;

в) параметры сборки: настройки компиляции, версия кода и имени приложения.

*– Жизненный цикл Activity:*

1) Жизненный цикл (Lifecycle) — это последовательность состояний, через которые проходит Activity от момента создания до момента уничтожения. Android система вызывает callback-методы (например, onCreate) при переходе между этими состояниями. Это позволяет вам управлять ресурсами (например, запускать и останавливать анимации, загружать и освобождать данные) в нужные моменты.

2) onCreate() – основной метод для первоначальной настройки Activity. Вызывается один раз при создании Activity.

2.1) Запускается при запуске приложения.

3) onStart() – Activity становится видимой пользователю, но еще не доступна для взаимодействия (не в фокусе).

3.1) Запускается после onCreate() или когда пользователь возвращается к приложению из фонового режима (onStop()).

4) onResume() – Activity выходит на передний план и становится готовой для взаимодействия с пользователем. Теперь она "активна".

4.1) Запускается после onStart() или когда поверх Activity исчезает другое окно (например, диалог или телефонный звонок).

5) onPause() – Activity частично перекрывается другим окном. Она теряет фокус, но остается частично видимой (например, когда открывается полупрозрачный диалог).

5.1) Запускается, когда система собирается перевести Activity в состояние onStop или onResume (если окно поверх исчезло).

6) onStop() – Activity становится полностью невидимой для пользователя.

6.1) Запускается, когда Activity полностью перекрывается другой Activity или пользователь сворачивает приложение.

7) onDestroy() – финализация Activity перед тем, как она будет уничтожена системой.

7.1) Запускается, когда Activity завершает работу (пользователь нажал "Назад") или система уничтожает ее для освобождения памяти.

*– Конфигурационные изменения:*

1) Поворот экрана — это конфигурационное изменение. По умолчанию система Android полностью уничтожает текущую Activity и создает ее заново с нуля, применяя альтернативные ресурсы (например, макет из папки layout-land).

1.1) Способы сохранения данных:

а) Использование ViewModel, который специально разработан для хранения и управления UI-данными. Он переживает конфигурационные изменения (например, поворот экрана). Данные остаются в памяти, и новая Activity получает доступ к той же самой ViewModel.

б) Сохранение состояния через onSaveInstanceState(Bundle) – это метод жизненного цикла, который система вызывает перед уничтожением Activity (например, при повороте). В него можно положить небольшие простые данные (строки, числа).

*– Навигация:*

1) Intent (намерение) — это механизм для выполнения операций, чаще всего для запуска другого компонента (Activity, Service и т.д.).

1.1) Явный Intent: точно указывает, какой компонент запустить (обычно для перехода внутри своего приложения).

// MainActivity.kt

val intent = Intent(this, SecondActivity::class.java)

startActivity(intent)

1.2) Неявный Intent: описывает действие, которое нужно выполнить (например, "отправить email"), и система сама находит подходящее приложение.

val intent = Intent(Intent.ACTION\_SEND).apply {

type = "text/plain"

putExtra(Intent.EXTRA\_EMAIL, arrayOf("test@example.com"))

putExtra(Intent.EXTRA\_SUBJECT, "Subject")

}

startActivity(intent)

2) Bundle — это контейнер (похожий на Map или словарь) для передачи данных типа "ключ-значение" между компонентами Android. Он может хранить примитивные типы и некоторые объекты, реализующие интерфейс Parcelable или Serializable.

2.1) Можно передать данные в новую Activity, положив их в Intent с помощью "extras", которые по сути являются Bundle.

val intent = Intent(this, SecondActivity::class.java)

intent.putExtra("USER\_NAME", "John Doe")

intent.putExtra("USER\_AGE", 30)

startActivity(intent)

2.2) Получение данных в SecondActivity:

class SecondActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_second)

val userName = intent.getStringExtra("USER\_NAME")

val userAge = intent.getIntExtra("USER\_AGE", 0)

// Используем данные...

}

}

Блок B: Пользовательский интерфейс (UI)

*– Иерархия View:*

1) View – это базовый строительный блок UI. Это отдельный элемент, который отображает информацию и обработка взаимодействия, и с которым может взаимодействовать пользователь.

2) ViewGroup – это невидимый контейнер, который содержит внутри себя другие View и дочерние ViewGroup. Он наследуется от View, но его главная задача — определять расположение дочерних элементов. Компоновка (layout) - управление размером и позицией дочерних View.

3) Ключевая разница:

а) View — это конечный элемент, который что-то показывает.

б) ViewGroup — это контейнер-невидимка, который организует порядок и расположение нескольких View на экране.

*– Основные элементы:*

1) TextView – это отображение текста на экране. Текст нельзя изменить пользователем (только для чтения). Ключевые свойства: text (текст для отображения), textSize (размер текста), textColor (цвет текста), gravity (выравнивание текста внутри самого TextView).

2) EditText – это поле для ввода и редактирования текста пользователем.

Ключевые свойства: (Наследует все от TextView) + специфические: hint (подсказка) и inputType (тип вводимых данных).

3) Button – это кнопка для обработки нажатий пользователя. Ключевые свойства: text (текст на кнопке), onClick (указание метода в коде, который вызывается при нажатии)

4) ImageView – это отображение картинок. Ключевые свойства: src/srcCompat (источник изображения), scaleType (определяет, как изображение будет масштабироваться и обрезаться для заполнения View).

*– Макеты:*

1) LinearLayout: располагает дочерние элементы последовательно, в одну линию — либо вертикально (vertical), либо горизонтально (horizontal).

1.1) Плюсы: очень простой и понятный.

1.2) Минусы: сложно создавать адаптивные и сложные интерфейсы. Для вложенности нескольких LinearLayout производительность падает.

2) FrameLayout: все дочерние элементы накладываются друг на друга, как стопка бумаг. Располагаются в левом верхнем углу.

2.1) Плюсы: идеален для отображения одного элемента или для ситуаций, когда элементы должны перекрываться (например, поверх основного контента показывается кнопка).

2.2) Минусы: не подходит для большинства сценариев компоновки.

3) ConstraintLayout: гибкая система, где положение и размер каждого элемента определяются ограничениями (constraints) относительно других элементов или родительского контейнера.

3.1) Преимущества:

а) гибкость и мощность: позволяет создавать любые, даже самые сложные макеты, без глубокой вложенности (flat hierarchy);

б) производительность: Меньшая вложенность (меньше ViewGroup внутри друг друга) приводит к лучшей производительности отрисовки;

в) адаптивность: макет легко адаптируется под разные размеры экранов, так как связи между элементами логические.

Вывод: ConstraintLayout является современным и рекомендуемым макетом по умолчанию для большинства задач из-за своей гибкости и производительности.

*– Ресурсы:*

1) Ресурсы хранятся в специальных папках внутри res/:

1.1) Строки: res/values/strings.xml: <string name="app\_name">Мое Приложение</string>

1.2) Цвета: res/values/colors.xml: <color name="purple\_500">#FF6200EE</color>

1.3) Размеры (Dimens): res/values/dimens.xml: <dimen name="standard\_margin">16dp</dimen>

1.4) Drawable (Изображения/фигуры): res/drawable/: сюда помещаются PNG, SVG файлы, а также XML-определения фигруг (например, кнопка с закругленными углами).

2) Почему важно выносить в ресурсы:

а) локализация: можно создать разные папки values-ru, values-en с переведенными строками;

б) поддержка разных конфигураций: можно иметь разные ресурсы для разных размеров экранов, ориентации, плотности пикселей;

в) cогласованность: централизованное хранение цветов, размеров и строк гарантирует, что во всем приложении используется единый стиль. Легко изменить цвет темы во всем приложении, поправив один файл;

г) разделение ответственности: код отвечает за логику, ресурсы — за внешний вид. Это делает проект чище и легче в поддержке.

Блок C: Работа со списками

*– RecyclerView архитектура:*

1) RecyclerView — это продвинутый и гибкий виджет для отображения больших наборов данных в виде прокручиваемого списка. Его архитектура основана на разделении ответственности между тремя основными компонентами:

1.1) Adapter – "мост" между данными и RecyclerView. Его главная задача — брать данные из источника (например, List) и "привязывать" их к элементам списка на экране.

1.2) ViewHolder кэширует ссылки на View элемента списка (например, на TextView и ImageView внутри одного элемента). Это предотвращает многократный дорогостоящий поиск View по ID (findViewById) при прокрутке.

1.3) LayoutManager отвечает за расположение элементов на экране. Определяет, в каком порядке и как элементы будут отображаться (линейно, сеткой, произвольно).

2) Как они взаимодействуют:

а) RecyclerView спрашивает у LayoutManager, сколько элементов помещается на экран;

б) RecyclerView просит Adapter создать необходимое количество ViewHolder'ов (onCreateViewHolder);

в) когда пользователь прокручивает список, RecyclerView сообщает Adapter'у: "Привяжи данные из позиции N к этому ViewHolder'у" (onBindViewHolder);

г) LayoutManager располаагет готовые ViewHolder'ы на экране;

в) при прокрутке ViewHolder'ы переиспользуются (recycle) для новых данных, что обеспечивает высокую производительность.

*– Оптимизация:*

1) Проблема: при обновлении данных в списке (например, после загрузки с сервера) простой вызов adapter.notifyDataSetChanged() заставляет RecyclerView полностью перерисовать все видимые элементы, даже если реально изменился всего один элемент. Это неэффективно и может привести к "морганию" интерфейса.

2) Решение: DiffUtil

2.1) Что это: утилита, которая вычисляет разницу между старым и новым списком данных.

2.2) Как помогает: она определяет минимальный набор операций (добавления, удаления, перемещения, изменения) для плавного и анимированного перехода от старого списка к новому.

2.3) Как используется:

а) создается класс, реализующий DiffUtil.Callback;

б) В нем определяются методы для сравнения элементов:

areItemsTheSame() — определяет, представляют ли два объекта один и тот же элемент (например, сравнивает ID);

areContentsTheSame() — определяет, совпадают ли данные внутри элемента (например, весь остальной контент).

в) результат (DiffUtil.DiffResult) передается адаптеру.

*–Взаимодействие:*

1) Подход 1: Обработка в Adapter / ViewHolder (Более простой)

Клик обрабатывается непосредственно внутри ViewHolder, который затем может уведомить внешний мир (например, Activity) через колбэк.

class MyViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {

// Объявляем View элемента

private val titleTextView: TextView = itemView.findViewById(R.id.title\_text\_view)

// Функция для привязки данных и установки слушателя

fun bind(item: MyItem, onItemClick: (MyItem) -> Unit) {

titleTextView.text = item.title

// Обработка клика на всем элементе списка

itemView.setOnClickListener {

onItemClick(item)

}

// Обработка клика на конкретной кнопке внутри элемента

itemView.findViewById<Button>(R.id.action\_button).setOnClickListener {

// Специфичное действие для кнопки

}

}

}

// В Activity/Fragment

val adapter = MyAdapter { clickedItem ->

// Реагируем на клик, например, открываем новый экран

val intent = Intent(this, DetailActivity::class.java)

intent.putExtra("ITEM\_ID", clickedItem.id)

startActivity(intent)

}

2) Подход 2: Делегирование через интерфейс (Более структурированный)

Создается интерфейс-слушатель, который реализуется Activity/Fragment.

// 1. Объявляем интерфейс

interface OnItemClickListener {

fun onItemClick(item: MyItem)

fun onActionButtonClick(item: MyItem)

}

// 2. Передаем реализацию в Adapter

class MyAdapter(private val listener: OnItemClickListener) : RecyclerView.Adapter<MyViewHolder>() {

...

override fun onBindViewHolder(holder: MyViewHolder, position: Int) {

holder.bind(dataList[position], listener)

}

}

// 3. Реализуем интерфейс в Activity/Fragment

class MyActivity : AppCompatActivity(), OnItemClickListener {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

// Передаем this (саму Activity) в качестве слушателя

recyclerView.adapter = MyAdapter(this)

}

override fun onItemClick(item: MyItem) {

// Обработка клика на элементе

}

override fun onActionButtonClick(item: MyItem) {

// Обработка клика на кнопке

}

}

Блок D: Работа с данными

*–SharedPreferences:*

1) SharedPreferences предназначены для хранения небольших объемов простых данных в формате ключ-значение. Это легковесное решение для сохранения настроек и простого состояния приложения.

2) Типичные use-cases:

2.1) настройки пользователя (тема приложения, язык, единицы измерения);

2.2) токены авторизации (но с осторожностью, не для критически важных данных);

2.3) флаги первого запуска приложения (onboarding);

2.4) простые счетчики или значения.

*–Room Database:*

1) Room — это надстройка над SQLite, которая предоставляет абстракцию для работы с базой данных, сокращая количество шаблонного кода и обеспечивая проверку на этапе компиляции.

2) Entity:

2.1) представляет таблицу в базе данных. Это обычный data-класс, аннотированный @Entity;

2.2) каждое поле класса — столбец в таблице.

3) DAO (Data Access Object):

3.1) интерфейс, который содержит методы для доступа к базе данных;

3.2) определяет SQL-запросы с помощью аннотаций (@Query, @Insert, @Update, @Delete).

4) Database:

4.1) абстрактный класс, расширяющий RoomDatabase;

4.2) содержит ссылки на все DAO и настройки базы данных.

5) Преимущества Room перед чистым SQLite:

а) проверка на этапе компиляции — SQL-запросы проверяются при компиляции, что исключает runtime-ошибки;

б) меньше шаблонного кода — не нужно писать CRUD-операции вручную;

в) интеграция с LiveData/Flow — автоматическое наблюдение за изменениями данных;

г) встроенная миграция схемы — удобные инструменты для обновления структуры БД;

д) type-safe запросы — компилятор проверяет соответствие типов данных.

*–Архитектурные паттерны:*

1) Repository Pattern – паттерн, который абстрагирует источник данных (источники данных) от остальной части приложения.

2) Как работает:

2.1) репозиторий предоставляет чистый API для операций с данными;

2.2) скрывает от бизнес-логики, откуда берутся данные (БД, сеть, кэш);

2.3) может комбинировать данные из нескольких источников.

3) Проблемы, которые решает:

3.1) разделение ответственности;

3.2) упрощение тестирования;

3.3) гибкость при смене источника данных.

4) ViewModel + LiveData/Flow проблемы, которые решают:

4.1) сохранение данных при смене конфигурации (поворот экрана): ViewModel автоматически сохраняется при повороте экрана, в отличие от Activity/Fragment;

4.2) Утечки памяти и управление жизненным циклом: ViewModel корректно очищается при окончательном уничтожении Activity/Fragment, а LiveData — lifecycle-aware компонент, который обновляет только активные подписчики

4.3) Реактивное программирование и автоматическое обновление UI: при изменении данных UI автоматически обновляется

1. ПРАКТИЧЕСКАЯ ЧАСТЬ

В листинге 1 представлен исправленный код.

Листинг 1 – Исправленный код

|  |
| --- |
| class ProblemActivity : AppCompatActivity() {    private var counter = 0  private lateinit var textView: TextView    override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  setContentView(R.layout.activity\_problem)    // Восстановление состояния  counter = savedInstanceState?.getInt("COUNTER\_KEY", 0) ?: 0    textView = findViewById(R.id.textView)  textView.text = "Счетчик: $counter"    // Исправленная ошибка 1  val handler = Handler(Looper.getMainLooper())  handler.postDelayed({  Toast.makeText(applicationContext, "Загрузка завершена", Toast.LENGTH\_SHORT).show()  }, 5000)    // Исправленная ошибка 2  loadHeavyData()    findViewById<Button>(R.id.incrementButton).setOnClickListener {  counter++  textView.text = "Счетчик: $counter"  }  }    private fun loadHeavyData() {  CoroutineScope(Dispatchers.IO).launch {  // Имитация тяжелой операции в фоновом потоке  delay(5000)  withContext(Dispatchers.Main) {  textView.text = "Данные загружены"  }  }  }    // Исправленная ошибка 3  override fun onSaveInstanceState(outState: Bundle) {  super.onSaveInstanceState(outState)  outState.putInt("COUNTER\_KEY", counter)  }  } |