МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ПОЛІТЕХНІКА»

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторних занять з дисципліни «Мобільно-орієнтована розробка програмного забезпечення»

Одеса: Одеська політехніка, 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ПОЛІТЕХНІКА»

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторних занять з дисципліни
«Мобільно-орієнтована розробка програмного забезпечення»
для здобувачів першого (бакалаврського) рівня вищої освіти

за спеціальністю

121 – Інженерія програмного забезпечення

Затверджено
на засіданні кафедри ІПЗ
Протокол № 1 від 30.08.2024 р.

Одеса: Одеська політехніка, 2025

Методичні рекомендації до лабораторних занять з дисципліни «Мобільно-орієнтована розробка програмного забезпечення» для здобувачів першого (бакалаврського) рівня вищої освіти за спеціальністю 121 – Інженерія програмного забезпечення (освітньо-професійна програма: Інженерія програмного забезпечення – 2024) / Укл.: Д. Р. Горпенко, О. М. Арнаутова. – Одеса: Одеська політехніка, 2025. – 25 с.

Укладачі: **Горпенко Д.Р.**, PhD, старший викладач
Арнаутова О. М., асистент

Методичні вказівки до лабораторних занять містять теоретичні відомості, які необхідні для роботи на лабораторних заняттях з дисципліни «Мобільно-орієнтована розробка програмного забезпечення».

Призначаються для студентів денної форми навчання.

ЗМІСТ

Вступ

У сучасному світі цифрових технологій успіх програмного продукту значною мірою залежить від його доступності та зручності для користувача. Мобільні застосунки стали ключовим інструментом взаємодії людини з інформаційними сервісами, бізнес-процесами та повсякденними завданнями. Дисципліна «Мобільно-орієнтована розробка програмного забезпечення» формує у студентів навички створення ефективних, надійних і зручних мобільних рішень, здатних працювати на різних платформах і відповідати сучасним вимогам до якості та продуктивності.

Метою практичних занять з дисципліни «Мобільно-орієнтована розробка програмного забезпечення» є формування у студентів навичок розуміння принципів мобільно-орієнтованої розробки, опанування вмінь розробляти сучасні, ефективні та інтерактивні мобільні застосунки для Android мовою Kotlin, а також здобуття знань щодо архітектури Android, принципів створення інтерфейсу користувача, роботи з даними, багатопоточності та інтеграції з мережею.

Лабораторне заняття 1

Визначення теми майбутнього застосунку. Створення першого застосунку та ознайомлення зі структурою проекту.

Мета заняття: навчити студентів визначати тему майбутнього мобільного застосунку, формулювати його основну ідею та цільову аудиторію, ознайомити зі структурою проекту Android-застосунку, а також засвоїти базові навички створення першого застосунку за допомогою Android Studio та мови програмування Kotlin.

Завдання: створення проєкту у Android Studio (Kotlin), налаштування AVD і запуск застосунку.

Теоретичні основи

Протягом курсу ми поступово будемо рухатися від базових понять до створення невеликого Android-застосунку. Завданням цього застосунку стане отримання даних з відкритого API, їх локальне збереження та візуалізація у зручному для користувача вигляді. На початковому етапі пропонується обрати ідею власного застосунку: тема повинна бути цікавою, відповідати чинному законодавству та етичним нормам, а також передбачати використання відкритого безкоштовного API. Серед доступних прикладів можна назвати сервіси погоди, курси валют, новини, інформацію про фільми, міську статистику, генератори анекдотів чи навіть базу знань про покемонів.

Для розробки Android-застосунків необхідні Android Studio як офіційне середовище розробки та Android SDK як набір інструментів і бібліотек. Рекомендується встановлювати ці компоненти на SSD-диск для швидшої роботи, а також мати щонайменше 4 ГБ оперативної пам’яті (оптимально більше). Процес інсталяції Android Studio простий: достатньо погодитися з ліцензійними умовами та обрати місце встановлення.

Після запуску Android Studio створення нового проєкту здійснюється через пункт меню **File → New Project**. Для першого застосунку доцільно обрати шаблон **Empty Views Activity**. На етапі налаштування задається назва застосунку, унікальний ідентифікатор пакету, мова програмування (рекомендовано Kotlin як основну) та мінімальна версія SDK. Мінімальна версія визначає найстарішу систему, на якій може працювати застосунок: чим нижчий API, тим більше пристроїв підтримується, але тим більше доведеться враховувати застарілі методи; чим вищий API, тим сучасніші інструменти доступні, проте зменшується кількість користувачів. Рекомендовано обирати такий рівень, що охоплює понад 90% пристроїв. Для керування збіркою використовується Gradle. Раніше застосовувався Groovy, зараз рекомендується Kotlin DSL. Gradle відповідає за завантаження бібліотек, конфігурації та складання застосунку, тому перша збірка проєкту може тривати довше і супроводжуватися тимчасовими повідомленнями про помилки.

Для тестування застосунків можна використовувати фізичний пристрій або віртуальний емулятор. Смартфон підключається через USB або Wi-Fi, що дозволяє одразу встановлювати застосунок. Проте для навчальних демонстрацій обов’язково потрібно налаштувати емулятор (AVD): обрати модель пристрою, завантажити версію Android, налаштувати параметри та запустити віртуальний пристрій. Це дозволяє перевіряти роботу застосунку на різних екранах і версіях системи.

Після створення проєкту й налаштування емулятора можна виконати перший запуск. Під час першого запуску Android Studio збирає застосунок, встановлює його на пристрій або

емулятор і виводить стандартне повідомлення **“Hello World”**. Це свідчить, що середовище працює коректно, і студент може переходити до подальшої розробки функціоналу. Таким чином, перші кроки в курсі включають обрання теми майбутнього застосунку та API, інсталяцію й налаштування Android Studio, створення нового проекту, конфігурацію емулятора і успішний запуск базового застосунку.

Завдання до лабораторної роботи:

1. Визначити тему майбутнього мобільного застосунку і обрати api, відповідно теми, яке ви будете використовувати в майбутньому.
2. Створити новий проект, вибравши потрібний шаблон.
3. Створити новий віртуальний пристрій у AVD Manager та запустити емулятор.
4. Запустити проект на емуляторі та переконатися, що застосунок відкривається без помилок. Додати невеликі зміни в проекті(замінити текст на ваше ПІБ та змінити колір фону) і виконати тестовий запуск застосунку, показавши, що зміни були враховані.
5. Підготовити звіт. У звіті описати обрану тему майбутнього застосунку, його ідею та цільову аудиторію.
6. Крок за кроком описати процес створення проекту в Android Studio.
7. Докладно описати процес створення емулятора та запуску застосунку.
8. Додати до звіту скріншоти:

а. Вікна створеного проекту в Android Studio.

б. Працюючого емулятора.

в. Працюючого застосунку на емуляторі.

Контрольні питання

1. Опишіть базову структуру Android-проєкту (папки java/, res/, manifests/, типи ресурсів).
2. Для чого використовується AVD Manager? Які параметри має віртуальний пристрій (ABI, роздільна здатність, версія Android)?
3. Які параметри задаються під час створення нового проєкту в Android Studio та чому важливо правильно обрати мінімальну версію SDK?

Лабораторне заняття 2

Розробка інтерфейсу користувача.

Мета заняття: навчити студентів проєктувати та створювати інтуїтивно зрозумілий інтерфейс користувача для Android-застосунків.

Зміст заняття: створити одновіконний Android-застосунок з інтерфейсом користувача; ознайомитися з базовими елементами взаємодії користувача; реалізувати логіку реакції на взаємодію з користувачем через обробку подій.

Теоретичні основи

У мобільній розробці під **Activity** розуміють умовне вікно застосунку. Кожна Activity складається з двох частин: класу-контролера та макету (layout). Хоча макет можна описати безпосередньо в коді, зазвичай використовується окремий XML-файл, пов’язаний із класом-контролером. Важливо, щоб назви класу й файлу макету узгоджувалися. Макети

створюються мовою XML і можуть редагуватися як у вигляді коду, так і через вбудований дизайнер.

У сучасній Android-розробці існують два підходи до створення інтерфейсу користувача: **Views** та **Jetpack Compose**. Compose - це новіший підхід, проте в навчальних цілях наразі доцільно розглядати класичні Views. У цьому підході ключовими є класи **View** (базовий для всіх елементів інтерфейсу) та **ViewGroup** (контейнер, що може містити інші елементи або групи). Усі елементи UI формують ієрархічне дерево. Надмірна кількість елементів ускладнює дерево і може знижувати продуктивність на слабких пристроях, тому важливо оптимізувати його структуру.

XML-файл є лише описом макету. Під час запуску його потрібно перетворити на об’єкти класів, що утворюють дерево інтерфейсу. Цей процес має назву **inflation**. Для цього використовується спеціальний клас **LayoutInflater**, метод inflate() якого перетворює опис макету у відповідні об’єкти. Наприклад, при створенні Activity викликається метод setContentView(R.layout.activity_main), що виконує інфляцію макету й приєднує його до вікна застосунку.

Кожному ресурсу Android присвоюється унікальний числовий ідентифікатор, який зберігається в автоматично згенерованому класі **R**. Це дозволяє звертатися до елементів інтерфейсу з коду. Класичний спосіб роботи з елементами UI передбачав використання методу findViewById, який шукає об’єкт у дереві за ідентифікатором. Наприклад:

```
val button = findViewById<Button>(R.id.button)
```

Цей метод повертає посилання на елемент, проте при великій кількості компонентів він може бути менш ефективним і створювати надлишковий код.

Щоб спростити роботу з елементами, у сучасних проєктах використовується **View Binding**. Ця технологія автоматично генерує спеціальний клас для кожного макету. Ім’я класу формується на основі назви макету з додаванням суфіксу *Binding*. Увімкнути View Binding можна в конфігураційному файлі build.gradle у розділі buildFeatures. Після цього стає можливим створювати об’єкт через метод inflate() і напряму звертатися до елементів за їхніми ідентифікаторами, без пошуку по дереву.

Наприклад, для макету activity_main.xml генерується клас ActivityMainBinding. З ним робота виглядає так:

```
val binding = ActivityMainBinding.inflate(layoutInflater)
setContentView(binding.root)
binding.button.setOnClickListener {
   binding.textView.text = "Clicked!"
}
```

Подібний підхід значно скорочує код, підвищує його читабельність і знижує ризик помилок. Логіка взаємодії з елементами UI реалізується через **події** (events) та **слухачі** (listeners). Наприклад, для кнопки типовою подією є натискання, яке обробляється за допомогою методу setOnClickListener. При виникненні події виконується прив’язаний до неї метод, що дозволяє реалізувати необхідну поведінку інтерфейсу.

Таким чином, робота з інтерфейсом у Android поєднує опис макету в XML, процес інфляції та використання сучасних підходів (наприклад, View Binding) для зручного доступу до елементів і організації взаємодії з користувачем.

Приклади можливих застосунків:

Конвертер одиниць:

- Введення значення (наприклад, метрів) і кнопка для конвертації у сантиметри або дюйми.
- Поле для відображення результату.

Симулятор кидання кубика:

- Кнопка для "кидання" кубика.
- Поле для випадкового числа від 1 до n відображається на екрані (кількість граней n оберіть на власний смак).

Простий блокнот:

- Поле введення для тексту.
- Кнопка для очищення тексту або збереження введеного тексту на екрані.

Завдання до лабораторної роботи:

1. Ознайомитися з наведеними прикладами можливих варіантів застосунків та обрати один із запропонованих варіантів.
2. Створити новий проект, реалізувати графічний інтерфейс користувача, використовуючи необхідні доступні елементи (Button, TextView, EditText...).
3. Написати логіку взаємодії у файлі MainActivity.kt.
4. Перевірити коректність роботи елементів взаємодії та логіки застосунку.
5. Підготовити звіт. У звіті описати:
   а. мету застосунку;
   б. реалізацію інтерфейсу користувача;
   в. реалізацію логіки взаємодії у Kotlin;
   г. додати скріншоти інтерфейсу застосунку та його роботи.

Контрольні питання

1. Що таке інтерфейс користувача (UI) і які його основні принципи (інтуїтивність, зрозумілість, простота)?
2. У чому різниця між TextView та EditText?
3. Як у Android реалізується обробка подій користувача (натискання кнопок, введення тексту)?
4. Які основні способи підключення обробників подій можно виділити?
5. Як організовано зв’язок між XML-розміткою інтерфейсу та кодом у MainActivity.kt?
6. Як у Kotlin реалізувати генерацію випадкового числа для симулятора кидання кубика?
7. Як реалізувати збереження та очищення введеного тексту у простому блокноті?
8. Як перевірити коректність роботи інтерфейсу та його логіки на емуляторі?
9. Чому важливо дотримуватися принципу «розділення логіки та інтерфейсу» у розробці застосунків?
10. Чому важливо тестувати UI як на емуляторі, так і на реальному пристрої?

Лабораторне заняття 3

Робота з колекціями та різними способами відображення їх у інтерфейсі користувача.

Мета заняття: навчити студентів працювати з колекціями даних у Kotlin, опрацьовувати та структурувати їх для відображення в Android-застосунках, ознайомити з різними способами візуалізації даних у користувацькому інтерфейсі.

Зміст заняття: створити Android-застосунок із декількома Activity; ознайомитись зі способом передачі даних між Activity; реалізувати логіку відкриття нового Activity та його закриття.

Теоретичні основи

У мові Kotlin **діапазони** дозволяють представляти інтервали значень. Для їх створення використовується оператор ... Наприклад, val range = 1..5 створює діапазон чисел від 1 до 5 включно, а val range1 = 'a'..'d' - діапазон символів. Діапазони можуть бути не лише числовими, а й символьними. Для зворотного порядку використовується функція downTo, а для пропуску верхньої межі - функція until. Крім того, можна задавати кроки між елементами за допомогою step. Для перевірки наявності елемента у діапазоні застосовується оператор in.

Масиви в Kotlin представлені типом Array. Визначаючи масив, у кутових дужках можна вказати тип елементів, наприклад: val numbers: Array<Int> = arrayOf(1, 2, 3, 4, 5). Функція arrayOf() дозволяє створювати масив із конкретними значеннями, а arrayOfNulls() - масив певної довжини з початковими null. Також можна використовувати конструктор Array(), передаючи розмір і лямбда-вираз для заповнення елементів. Масиви можуть містити будь-які типи, зокрема Any або Any?, що дозволяє зберігати різнорідні дані, у тому числі й null.

Доступ до елементів масиву здійснюється за індексом, як у Java: numbers[1] повертає другий елемент, а присвоєння numbers[2] = 7 змінює третій елемент. Кожен масив має властивість size, що повертає кількість його елементів. Для перебору елементів можна використовувати цикл for. Крім того, оператори in та !in дозволяють перевіряти наявність або відсутність значень у масиві.

Окрім універсального типу Array, у Kotlin існують спеціалізовані масиви для примітивних типів: BooleanArray, IntArray, DoubleArray тощо. Вони створюються функціями intArrayOf(), doubleArrayOf() та аналогічними конструкторами, наприклад: val doubles: DoubleArray = doubleArrayOf(2.4, 4.5, 1.2) або val doubles1 = DoubleArray(3) { 1.5 }.

У мові Kotlin для роботи з колекціями, крім масивів, активно використовуються **List**, **Set** та **Map** разом із їхніми різними реалізаціями.

List - це впорядкована колекція, яка може містити повторювані елементи. Списки бувають незмінними (listOf) та змінними (mutableListOf). Доступ до елементів здійснюється за індексом, починаючи з нуля. Наприклад:

```
val numbers = listOf(1, 2, 3, 4)
val mutableNumbers = mutableListOf(1, 2, 3)
println(numbers[0])     // 1
mutableNumbers.add(4)   // [1, 2, 3, 4]
mutableNumbers.remove(2) // [1, 3, 4]
```

Set - це колекція унікальних елементів без гарантованого порядку. Якщо додати дублікати, вони автоматично відкидаються. Використовуються setOf для незмінних і mutableSetOf для змінних множин. Наприклад:

```
val items = setOf(1, 2, 3, 3, 4)    // результат: [1, 2, 3, 4]
val mutableItems = mutableSetOf("a", "b")
mutableItems.add("c")   // ["a", "b", "c"]
mutableItems.remove("a") // ["b", "c"]
```

Map - це колекція пар ключ–значення. Ключі завжди унікальні, тоді як значення можуть повторюватися. Для створення використовують mapOf або mutableMapOf. Приклад:

```
val users = mapOf(1 to "Tom", 2 to "Sam")
val mutableUsers = mutableMapOf(1 to "Tom")
println(users[1])          // Tom
mutableUsers[2] = "Bob"    // {1=Tom, 2=Bob}
mutableUsers.remove(1)     // {2=Bob}
```

HashSet - реалізація множини, яка зберігає тільки унікальні значення, але не гарантує порядку.

```
val hashSet = hashSetOf(5, 3, 8, 1, 3) // [5, 3, 8, 1]
hashSet.add(10)
hashSet.remove(3)
```

HashMap - реалізація Map на основі хеш-таблиць, яка також не гарантує порядок ключів.

```
val hashMap = hashMapOf(1 to "A", 2 to "B")
hashMap[3] = "C"          // {1=A, 2=B, 3=C}
hashMap.remove(1)         // {2=B, 3=C}
```

LinkedHashSet - множина, яка зберігає унікальні елементи, але при цьому пам’ятає порядок їх додавання.

```
val linkedSet = linkedSetOf(3, 1, 2, 1) // [3, 1, 2]
```

LinkedHashMap - карта, яка зберігає порядок додавання ключів.

```
val linkedMap = linkedMapOf(1 to "One", 2 to "Two")
linkedMap[3] = "Three"    // {1=One, 2=Two, 3=Three}
```

Таким чином, використання List, Set, Map та їхніх реалізацій дозволяє будувати колекції з потрібними властивостями: впорядкованістю, унікальністю чи швидким доступом до даних.

LinearLayout - це один із базових контейнерів у Android, який використовується для розташування елементів інтерфейсу користувача. Особливістю LinearLayout є те, що всі його дочірні елементи розташовуються послідовно в одному напрямку - або **вертикально** (зверху вниз), або **горизонтально** (зліва направо). Напрямок задається атрибутом android:orientation, значенням якого може бути "vertical" або "horizontal".

Кожен елемент усередині LinearLayout може мати власні параметри вирівнювання, відступів та ваги. Атрибут layout_weight дозволяє розподіляти простір між елементами. Наприклад, якщо одному елементу задано вагу 1, а іншому 2, то другий елемент отримає удвічі більше місця.

Для відображення колекцій LinearLayout можна використовувати як основу, розташовуючи елементи в потрібному порядку. Наприклад, якщо потрібно вивести список значень, для кожного елемента можна програмно створювати TextView і додавати його до LinearLayout у циклі. Це зручно для невеликих наборів даних, коли повноцінні компоненти на зразок RecyclerView ще не потрібні.

Приклад додавання елементів у коді Kotlin:

```
val linearLayout = findViewById<LinearLayout>(R.id.linearLayout)
val items = listOf("Tom", "Sam", "Bob")

for (item in items) {
    val textView = TextView(this)
    textView.text = item
    linearLayout.addView(textView)
}
```

Приклади модифікацій застосунків, які були обрані в лабораторній роботі 2:

Конвертер одиниць:

- Додати додатково декілька форматів конвертації(наприклад: м→см, і тд.).
- Головне Activity дозволяє обрати потрібний формат конвертації, вводити значення та конвертувати їх.
- Друге Activity отримує нове значення і відображає його (історію конвертацій)

Симулятор кидання кубика:

- Головне Activity містить кнопку для генерації випадкового числа (1–n).

- Друге Activity отримує нове значення і відображає його (історію кидків)

Простий блокнот:

- Головне Activity дозволяє вводити та зберігати рядок.
- Друге Activity отримує новий рядок і відображає його (список збережених переданих до другої Activity записів).

Завдання до лабораторної роботи:

1. Обрати реалізований застосунок з попереднього лабораторного заняття.
2. Розширити його функціонал, додавши друге Activity.
   а. Додати нове Activity до проєкту;
   б. Налаштувати перехід між Activity за допомогою Intent;
   в. Реалізувати передачу даних між Activity;
   г. Реалізувати кнопку для повернення до головного екрану.
3. Підготовити звіт. У звіті описати:
   а. мету застосунку;
   б. реалізацію інтерфейсу користувача;
   в. реалізацію логіки взаємодії у Kotlin;
   г. додати скріншоти інтерфейсу застосунку та його роботи.

Контрольні питання

11. Що таке Activity в Android? Які основні методи її життєвого циклу?
12. Чим відрізняється List від Set у Kotlin?
13. Що таке HashSet і чим він відрізняється від LinkedHashSet?
14. Які класи в Kotlin використовуються для генерації випадкових чисел (Random)? Як вони застосовуються у симуляторі кубика?
15. Як забезпечити, щоб дані відображалися у вигляді списку в Activity?
16. Як можна організувати збереження історії введених даних (наприклад, у блокноті) після закриття застосунку?
17. Як протестувати роботу передачі даних між Activity на емуляторі?

Що таке HashSet і чим він відрізняється від LinkedHashSet?

Лабораторне заняття 4

Створення багатовіконного застосунку. Взаємодія між вікнами. Зберігання даних.

Мета заняття: навчити студентів створювати багатовіконні Android-застосунки, організовувати взаємодію між вікнами, передавати та отримувати дані, а також опанувати базові методи зберігання даних.

Зміст заняття: використання основних віджетів при побудові інтерфейсу користувача; розробка першого макету інтерфейсу мобільного застосунку відповідно до обраної теми; налагодження навігації між вікнами застосунку з передачею та поверненням даних; ознайомлення та використання з View Binding.

Теоретичні основи

У процесі розробки Android-застосунків часто виникає потреба організувати перехід між різними вікнами (Activity). Наприклад, користувач натискає кнопку, переходить до нового вікна, виконує там певні дії й потім повертається назад. Для цього у середовищі

Android Studio ми можемо створити нову Activity: у меню обираємо **File → New → Activity → Empty Views Activity**, надаємо їй назву, після чого в проєкті з'являється нове вікно, що складається з XML-макету та Kotlin-файлу. Крім того, у файлі AndroidManifest.xml автоматично додається запис про цю Activity. Рекомендується створювати нові екрани саме через майстер, щоб уникнути помилок у конфігурації, адже при ручному додаванні можна випадково пропустити важливі елементи й отримати помилки під час запуску.

Щоб організувати перехід із одного вікна до іншого, потрібно додати кнопку, натискання якої відкриватиме нову Activity. В Android для цього використовується спеціальний механізм під назвою **Intent** (з англійської - «намі́р»). Intent дозволяє повідомити системі, яку дію потрібно виконати, і при цьому може використовуватися для передачі даних між компонентами застосунку. Для створення нового наміру необхідно оголосити об'єкт Intent, передати йому контекст поточного вікна та клас Activity, яку потрібно відкрити, і викликати метод startActivity(intent). Приклад:

```
view.button.setOnClickListener {
   val intent = Intent(this, SecondActivity::class.java)
   startActivity(intent)
}
```

Тут this означає контекст поточного вікна, а другим параметром вказується клас Activity, що має бути відкритий.

Існує два типи намірів: **явні** та **неявні**. Явний Intent використовується тоді, коли ми точно вказуємо клас Activity, яку потрібно відкрити. Неявний Intent застосовується для більш абстрактних дій - наприклад, «відкрити веб-сторінку», «поділитися зображенням» або «запустити камеру». У цьому випадку Android сам пропонує користувачу список застосунків, які можуть виконати дію, і користувач обирає потрібний. Така поведінка ґрунтується на тому, що в системі зберігається інформація про те, які Activity здатні обробляти конкретні інтенти.

Intent також може містити додаткові дані. Це реалізується за допомогою методу putExtra(), у якому вказується ключ і значення. Наприклад, можна передати текст із одного вікна в інше:

```
intent.putExtra("text_key", view.textView1.text.toString())
startActivity(intent)
```

У другій Activity ці дані можна зчитати методом getStringExtra(). Використовуючи ViewBinding, це робиться так:

```
val view = ActivitySecondBinding.inflate(layoutInflater)
setContentView(view.root)
val receivedText = intent.getStringExtra("text_key")
view.textView.text = receivedText
```

Таким чином отриманий текст одразу буде відображений у новому вікні.

Щоб реалізувати повернення з другої Activity назад, достатньо додати кнопку, якій прив'язати слухач події OnClickListener, що викликатиме метод finish(). Виклик finish() завершує роботу поточної Activity та повертає користувача до попереднього вікна.

Отже, робота з Activity у Kotlin передбачає створення нових екранів через Android Studio, використання механізму Intent для переходів і передачі даних, а також застосування ViewBinding для зручної роботи з інтерфейсом. Це дозволяє організовувати логіку взаємодії між вікнами застосунку та забезпечувати плавний сценарій роботи користувача.

У процесі розробки Android-застосунків часто виникає потреба передавати між Activity не лише прості дані (рядки чи числа), а й складніші об’єкти. Для цього використовується механізм **серіалізації**. Стандартна Java надає інтерфейс Serializable, однак у середовищі Android більш ефективним є інтерфейс **Parcelable**, який спеціально оптимізований для мобільних пристроїв і є рідною частиною Android SDK.

Використання Serializable у Android можливе, але воно не оптимізоване й призводить до зайвого навантаження на пам’ять та збільшення розміру даних, що передаються. Натомість Parcelable працює швидше, бо безпосередньо пакує об’єкти в байти й дозволяє передавати їх через Intent або зберігати під час зміни стану. Тому при створенні власних класів, які потрібно передавати між Activity, варто відразу реалізовувати інтерфейс Parcelable.

У Kotlin найзручніше реалізувати Parcelable за допомогою анотації @Parcelize. Для цього необхідно підключити плагін у build.gradle (Module) через id("kotlin-parcelize"), після чого досить позначити клас анотацією @Parcelize. Система автоматично згенерує потрібний код для серіалізації та десеріалізації. Усередині механізму працюють методи: writeToParcel, який записує дані до об’єкта Parcel, та createFromParcel, який відновлює об’єкт із тих самих даних у тому самому порядку. Таким чином можна передати, наприклад, об’єкт класу User із однієї Activity до іншої без додаткових складнощів.

Крім передачі даних у нову Activity, іноді виникає потреба не лише відкрити новий екран, а й отримати результат його роботи назад. Для цього у сучасному Android використовується механізм **ActivityResultLauncher**. Його реєстрація відбувається в основній Activity за допомогою методу registerForActivityResult. Цей метод створює обробник, який «чекає» на результат виконання іншої Activity. Якщо повертається код RESULT_OK, то можна отримати передані дані через Intent і виконати необхідні дії, наприклад показати повідомлення або оновити інтерфейс.

Процес виглядає так: спочатку у MainActivity реєструється лаунчер, який слухає результат. Потім за допомогою методу launch(intent) відкривається нова Activity, і основна Activity залишається у стеку, очікуючи на відповідь. У відкритій Activity, після виконання потрібних дій, формується Intent із результатом (наприклад, текстовим повідомленням), який повертається викликом setResult(RESULT_OK, resultIntent). Виклик finish() закриває Activity і повертає керування до попереднього вікна. Після цього обробник результату у MainActivity отримує дані та може їх використати - наприклад, показати повідомлення за допомогою Toast.

Таким чином, у Android є два важливих механізми для роботи з даними між Activity: **Parcelable** для ефективної передачі складних об’єктів і **ActivityResultLauncher** для повернення результатів роботи нової Activity у попередню. Використання цих механізмів дозволяє будувати більш гнучку й логічно завершену взаємодію між екранами застосунку.

Неявний інтент в Android - це механізм, який дозволяє нашому застосунку запускати інші програми на пристрої для виконання певної дії. При цьому ми не вказуємо конкретний додаток, що має бути відкритий, а лише описуємо, яку саме операцію необхідно виконати. Система Android сама знаходить усі програми, здатні обробити цей запит, і надає користувачу можливість обрати потрібну.

Приклади використання неявних інтентів - це відкриття веб-сторінки у браузері, здійснення телефонного дзвінка чи надсилання електронного листа. Усі застосунки, які можуть виконувати подібні дії, попередньо реєструють у своєму файлі **AndroidManifest.xml**

так звані *фільтри інтентів*. Завдяки цим фільтрам система розуміє, який застосунок здатний обробити певний інтент.

Коли ми створюємо неявний інтент, наприклад із дією ACTION_SEND та типом message/rfc822 (що означає надсилання повідомлення електронною поштою), Android виконує кілька кроків. Спочатку система перевіряє всі встановлені програми, здатні виконати цю дію. Якщо відповідний застосунок знайдено, він відкривається або користувачу пропонується список програм для вибору. Якщо жоден додаток не може виконати потрібну дію, метод resolveActivity() повертає null, що означає відсутність доступного обробника.

Таким чином, неявні інтенти дозволяють створювати універсальні сценарії взаємодії між додатками, роблячи застосунок більш інтегрованим у середовище Android та зручним для користувача.

Завдання до лабораторної роботи:

1. Створити макет основних вікон застосунку:
   а. Створити необхідну кількість вікон;
   б. Обов'язково використати компоненти інтерфейсу: TextView, EditText, Button, Checkbox, RadioButton, SeekBar;
   в. Розмістити елементи логічно відповідно до обраної теми застосунку (наприклад, якщо це застосунок для підрахунку калорій - поля для введення їжі, вибір категорії, чекбокси для додаткових опцій тощо).
2. Створити Activity, що буде повертати результат:
   а. Створити окреме Activity, що відкриватиметься для певної дії і яке буде повертати результат;
   б. Передавати в це Activity дані з викликаючого Activity через Intent;
   в. Після виконання певних дій у викликаному Activity - повернути результат назад у попереднє Activity;
   г. Закрити викликане Activity після передачі результату.
3. Налаштувати навігацію між вікнами і описати її
4. Реалізувати використання View Binding
5. Додати логіку відображення повідомлень для користувача через Toast або Snackbar
6. Підготовити звіт. У звіті описати:
   а. тему застосунку;
   б. створений макет інтерфейсу;
   в. навігацію між вікнами - логіку переходів між Activity;
   г. використання View Binding;
   д. додати скріншоти вікон застосунку.

Контрольні питання

1. Що таке багатовіконний застосунок в Android і для чого його використовують?
2. Які основні способи навігації між Activity ви знаєте?
3. У чому відмінність між явним та неявним Intent?
4. Як передати дані з одного Activity до іншого за допомогою Intent?
5. Які методи використовуються для повернення результату з Activity?
6. Які компоненти інтерфейсу використовуються для введення та вибору даних

користувачем?

7. У чому різниця між CheckBox і RadioButton? Наведіть приклади використання.
8. Що таке View Binding і які його переваги над findViewById?
9. Як організувати логіку закриття Activity після передачі результату?
10. Як організувати логіку перевірки правильності введених даних користувачем у EditText?
11. Які можливі помилки при передачі даних між Activity (null-значення, неправильний ключ, типи даних)?

Лабораторне заняття 5

Багатопоточність.

Мета заняття: навчити студентів основам багатопоточності в Android-застосунках, ознайомити з основними підходами до створення та управління потоками, забезпечити розуміння принципів асинхронного виконання завдань для підвищення продуктивності та уникнення блокування основного потоку користувацького інтерфейсу.

Зміст заняття: використовання RecyclerView для створення та відображення списків; освоєння роботи з фрагментами та їх інтеграцію в за стосунках; забезпечення передачі даних у фрагменти та між ними.

Теоретичні основи

У мові Kotlin підтримка асинхронного виконання та паралельних обчислень реалізована за допомогою **корутин** (*coroutines*). Корутина - це блок коду, який може виконуватися паралельно з іншим кодом, не блокуючи основний потік. В Android це використовується для того, щоб виконувати тривалі операції (наприклад, завантаження даних з інтернету) без «підвисання» інтерфейсу користувача. Базова функціональність корутин зосереджена в бібліотеці **kotlinx.coroutines**, яку потрібно підключати додатково, оскільки вона за замовчуванням у проєкті не встановлена.

Щоб додати бібліотеку корутин, у середовищі Android Studio необхідно відкрити **Project Structure → Libraries**, додати нову залежність через пункт **From Maven…** та знайти kotlinx-coroutines-core-jvm. Після цього бібліотека з’явиться серед ресурсів проєкту. Далі у файлі build.gradle.kts у розділі залежностей додається рядок:

implementation(libs.kotlinx.coroutines.core)

Для використання корутин потрібно оголошувати їх у межах спеціальної області — **coroutine scope**. Область корутин визначає життєвий цикл усіх корутин, створених у ній, і завершується лише тоді, коли завершуються всі вкладені корутини. Приклад простої корутини:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope {
    launch {
        for (i in 0..5) {
            delay(400L)
            println(i)
        }
    }
    println("Hello Coroutines")
```

}

У цьому прикладі функція coroutineScope створює область корутин, у межах якої за допомогою функції launch запускається корутина. Вона виконує цикл, що затримується на 400 мс на кожній ітерації. При цьому головний потік не блокується: рядок "Hello Coroutines" може бути виведений незалежно від виконання циклу. Оператор delay() призупиняє корутину, звільняючи потік, і дозволяє системі виконувати інші завдання. Після завершення затримки корутина відновлює свою роботу.

Корутини можуть виконуватися паралельно. Наприклад, у межах однієї функції можна запустити кілька корутин одночасно, і область coroutineScope чекатиме завершення всіх них. Важливо розуміти, що функція runBlocking блокує головний потік до моменту, поки всі корутини всередині не завершаться.

Для запуску корутин у Kotlin існують два основні побудовники: **launch** та **async**.

- launch використовується тоді, коли результат виконання корутини не потрібен. Воно повертає об’єкт типу **Job**, за допомогою якого можна керувати корутиною (наприклад, скасувати її викликом cancel()).
- async застосовується тоді, коли потрібно отримати результат. Воно повертає об’єкт типу **Deferred**, який є підтипом Job і додатково дозволяє дочекатися результату виконання корутини за допомогою методу await().

Таким чином, корутини є сучасним і ефективним механізмом асинхронного програмування у Kotlin та Android. Вони дозволяють виконувати тривалі обчислення або операції вводу-виводу без блокування головного потоку, роблячи застосунок більш продуктивним і зручним для користувача.

Завдання до лабораторної роботи:

1. Оптимізація застосунку з попередньої лабораторної роботи:
   а. Замість частини зайвих **Activity**, де це можливо, використати **фрагменти**;
   б. Якщо заміна не є доцільною — додати хоча б **номінальне використання фрагменту**, щоб обов'язково продемонструвати вміння працювати з ними.
2. Створення та використання списку:
   а. Додати до застосунку **список** (наприклад, список попередньо введених користувачем даних);
   б. Реалізувати передачу даних для заповнення списку;
   в. Реалізувати можливість вибрати елемент зі списку.
3. Фрагменти та взаємодія з ними:
   а. Фрагменти повинні **отримувати дані** та **відображати їх на екрані;**
   б. *додаткове завдання - реалізувати **передачу даних з фрагментів в інші фрагменти або активності** (наприклад, через інтерфейси або ViewModel);
   в. Організувати навігацію між фрагментами.
4. Для відображення повідомлень використовувати Toast або Snackbar;
5. Усі елементи інтерфейсу розмістити логічно та відповідно до тематики застосунку;
6. Підготовити звіт. У звіті описати:
   а. тему застосунку;
   б. основні зміни у порівнянні з попередньою лабораторною роботою;
   в. як були інтегровані фрагменти та списки;
   г. логіку навігації та передачі даних;

д. додати скріншоти фрагментів та списків.

Контрольні питання

1. Що таке багатопоточність і чому вона важлива в Android-застосунках?
2. Які ризики виникають при виконанні тривалих операцій у головному (UI) потоці?
3. Чим відрізняється синхронне та асинхронне виконання завдань?
4. Що таке корутина і чим вона відрізняється від звичайного потоку (thread)?
5. Для чого використовується бібліотека **kotlinx.coroutines** і як її підключити до проєкту?
6. Що таке **coroutine scope** і яку роль він відіграє у виконанні корутин?
7. Яку функцію виконує ключове слово **suspend** у Kotlin?
8. Для чого використовується функція **launch** і який об’єкт вона повертає?
9. У чому різниця між **launch** та **async** у корутинах?
10. Що таке об’єкт **Deferred** і як отримати з нього результат виконання корутини?
11. Яку роль відіграє функція **delay()** у корутині і чим вона відрізняється від Thread.sleep()?
12. Як можна скасувати виконання корутини і який об’єкт при цьому використовується?

Лабораторне заняття 6

Методи роботи з мережею інтернет.

Мета заняття: навчити студентів використовувати методи роботи з мережею інтернет в Android-застосунках, ознайомити з базовими концепціями HTTP-запитів, обробкою API-відповідей у форматах JSON та XML.

Зміст заняття: Ознайомлення з бібліотеками Retrofit2 та OkHttp для роботи з HTTP-запитами; навчання здійснювати запити до REST API та обробляти відповіді; інтегрування отримані дані у вже створений застосунок.

Теоретичні основи

REST API (*Representational State Transfer Application Programming Interface*) - це архітектурний стиль, який визначає правила взаємодії клієнта та сервера через протокол HTTP. Він дозволяє клієнтським застосункам виконувати стандартні операції над ресурсами: отримувати, створювати, змінювати або видаляти дані. Типовий сценарій роботи виглядає так: клієнт (наприклад, вебсторінка з JavaScript) надсилає HTTP-запит на сервер, що містить дані (наприклад, ідентифікатор товару та кількість). Сервер обробляє запит за правилами REST, виконує необхідні дії з базою даних і повертає відповідь. У результаті клієнт отримує оновлені дані або повідомлення про помилку й відображає зміни в інтерфейсі користувача.

Для роботи з REST API в Android зручно використовувати бібліотеку **Retrofit**, створену компанією *Square*. Вона дозволяє описувати API у вигляді інтерфейсів, а потім автоматично формує HTTP-запити на їх основі. Retrofit підтримує JSON-конвертацію (за допомогою Gson або Moshi), інтегрується з Kotlin Coroutines і повертає відповіді у вигляді об’єктів Kotlin/Java. Наприклад:

```
interface ApiService {
  @GET("users/{id}")
```

```
    suspend fun getUser(@Path("id") id: Int): User
}
```

Основна ідея Retrofit полягає в тому, що розробник описує API через інтерфейс із анотаціями (@GET, @POST тощо), налаштовує базовий URL і конвертер, а бібліотека сама формує запит і перетворює JSON-відповідь у Kotlin-об’єкти. Всі запити під капотом виконує низькорівнева бібліотека **OkHttp**, яка є «двигуном» мережевої взаємодії.

OkHttp — це HTTP-бібліотека, що підтримує сучасні протоколи (HTTP/2, WebSocket, TLS), дозволяє додавати інтерцептори для логування або обробки токенів, виконує кешування відповідей і працює асинхронно. Вона надає гнучкі можливості керування таймаутами, проксі та редиректами. Основними компонентами OkHttp є об’єкти Request (опис запиту), Interceptor (для зміни або логування запитів), Dispatcher (керування чергами), RealCall (обробка запиту) та Socket (з’єднання із сервером). Якщо Retrofit виступає «фасадом», що описує логіку, то OkHttp є «двигуном», який реально виконує запити.

Для доступу до мережі Android-застосунок повинен мати дозвіл. У файлі **AndroidManifest.xml** обов’язково потрібно вказати:

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

Це повідомляє систему, що застосунок має право використовувати інтернет-з’єднання. Без цього рядка будь-які спроби звернення до мережі будуть заблоковані.

Підключення зовнішніх бібліотек відбувається у файлі **build.gradle (Module)** у блоці dependencies. Зокрема:

1. implementation("com.squareup.retrofit2:retrofit:2.9.0") - основна бібліотека Retrofit для роботи з REST API.
2. implementation("com.squareup.retrofit2:converter-gson:2.9.0") - модуль для автоматичної конвертації JSON ↔ Kotlin/Java об’єкти.
3. implementation("com.squareup.okhttp3:okhttp:4.12.0") - бібліотека OkHttp для виконання HTTP-запитів.
4. implementation("com.squareup.okhttp3:logging-interceptor:4.12.0") - інтерцептор для логування усіх HTTP-запитів та відповідей у Logcat.
5. implementation(libs.squareup.picasso) - бібліотека Picasso для завантаження та кешування зображень у ImageView.

Налаштування Retrofit зазвичай відбувається в окремому методі, наприклад initRetrofit():

```
val interceptor = HttpLoggingInterceptor().apply {
    level = HttpLoggingInterceptor.Level.BODY
}
val client = OkHttpClient.Builder().addInterceptor(interceptor).build()
val retrofit = Retrofit.Builder()
    .baseUrl("https://dummyjson.com")
    .client(client)
    .addConverterFactory(GsonConverterFactory.create())
    .build()

mainApi = retrofit.create(MainApi::class.java)
```

У цьому прикладі створюється логуючий інтерцептор, додається клієнт OkHttp, підключається конвертер Gson для перетворення JSON у Kotlin-об’єкти, а також формується інтерфейс MainApi для роботи з методами API.

Завдання до лабораторної роботи:

1. Підготовка застосунку:
   а. Відкрити застосунок, над яким працювали в попередніх лабораторних роботах;
   б. За потреби, оптимізувати структуру або оновити дизайн.
2. Робота з мережею:
   а. Обрати API, яке було вибране в попередніх роботах або вибрати нове;
   б. Створити клас API-сервісу з використанням бібліотеки Retrofit2;
   в. Налаштувати OkHttpClient для логування запитів/відповідей.
   г. Реалізувати модель даних, яка відповідає відповіді API
   д. Отримати дані з API та відобразити їх у застосунку (наприклад, список товарів, постів, профілів тощо)
3. Інтеграція в інтерфейс:
   а. Дані, отримані з API, повинні бути відображені (наприклад, за допомогою RecyclerView);
   б. Забезпечити можливість перегляду детальної інформації про вибраний елемент (через фрагмент або окремий екран);
   в. Забезпечити обробку помилок запиту (відображення повідомлення про помилку через Toast або Snackbar).
4. Навігація та UX:
   а. Реалізувати логічну навігацію між фрагментами або екранами;
5. Підготовити звіт. У звіті описати:
   а. Тему застосунку та обране API;
   б. Як було реалізовано запит до API та які бібліотеки для цього використовувалися;
   в. Як була здійснена інтеграція отриманих даних у застосунок;
   г. Логіку навігації та оновлення інтерфейсу;
   д. Скріншоти основних фрагментів застосунку після інтеграції мережевої функціональності.

Контрольні питання

1. Що таке REST API і які принципи його побудови?
2. Які типові коди відповідей HTTP ви знаєте (200, 201, 400, 401, 403, 404, 500)?
3. Для чого використовується бібліотека Retrofit2?
4. Як у Retrofit описуються методи API та яку роль відіграють анотації (@GET, @POST, @Path, @Query)
5. Яку роль виконує OkHttp у мережевих запитах?
6. Що таке OkHttpClient і як його можна налаштувати (таймаути, логування)?
7. Яке призначення має HttpLoggingInterceptor та в яких випадках його доцільно використовувати?

Лабораторне заняття 7

Налагодження роботи WorkManager.

Мета завдання: навчити студентів використовувати компонент WorkManager для виконання фонових завдань в Android-застосунках, ознайомити з налаштуванням, запуском і моніторингом задач.

Зміст завдання: налаштування WorkManager для виконання фонового завдання, пов'язаного зі збереженням або синхронізацією даних; ознайомлення з механізмами збереження даних у Android-застосунках, включаючи використання бази даних (наприклад, Room) та SharedPreferences / EncryptedSharedPreferences; навчання інтегрувати механізми збереження даних у застосунок та обґрунтувати вибір конкретного способу збереження відповідно до задач застосунку;

Теоретичні основи

WorkManager - це компонент Android Jetpack, призначений для виконання відкладених та довготривалих фонових завдань, які мають виконуватись гарантовано, навіть якщо застосунок буде закрито або пристрій перезавантажиться. Це універсальний інструмент для сценаріїв, де важливо, щоб робота була завершена, наприклад: синхронізація даних з сервером, резервне копіювання або періодичне оновлення контенту.

WorkManager інтегрується з системними механізмами Android, зокрема JobScheduler, AlarmManager та BroadcastReceiver, автоматично обираючи оптимальний спосіб виконання залежно від версії Android і стану пристрою. Таким чином, розробнику не потрібно вручну писати складний код для сумісності з різними версіями ОС.

Основна ідея WorkManager полягає в описі роботи через клас **Worker**, у якому визначається логіка завдання. Кожна задача огортається в об'єкт **WorkRequest**, який передає інформацію про те, коли й за яких умов вона має бути виконана. Завдання можуть мати залежності, виконуватись одноразово (**OneTimeWorkRequest**) або періодично (**PeriodicWorkRequest**).

Для роботи з WorkManager у файлі **build.gradle (Module)** потрібно додати залежність:

```
implementation("androidx.work:work-runtime-ktx:2.9.0")
```

Приклад створення завдання:

```
class SyncWorker(appContext: Context, workerParams: WorkerParameters) :
    Worker(appContext, workerParams) {
    override fun doWork(): Result {
        // Логіка синхронізації
        return Result.success()
    }
}
```

Запуск однієї роботи:

```
val request = OneTimeWorkRequestBuilder<SyncWorker>().build()
WorkManager.getInstance(context).enqueue(request)
```

У WorkManager передбачено три результати виконання завдання:

- Result.success() - робота успішно завершена;
- Result.failure() - робота завершилася з помилкою;
- Result.retry() - робота має бути повторена пізніше.

Можливості WorkManager включають:

- встановлення умов виконання (наприклад, лише при підключенні до Wi-Fi або зарядному пристрою);
- побудову ланцюжків завдань, коли виконання одного залежить від завершення іншого;
- збереження результатів між завданнями через Data;
- спостереження за станом завдань у реальному часі.

У процесі розробки мобільних застосунків часто виникає потреба зберігати невеликі обсяги даних для подальшого використання. Це можуть бути налаштування користувача, параметри конфігурації чи інші допоміжні значення. В Android для цього використовується механізм **Preferences**, що реалізується у вигляді пар «ключ–значення». У класі Activity (через його батьківський клас Context) доступний метод getSharedPreferences(), за допомогою якого можна створювати або зчитувати налаштування. Наприклад:

```
val preferences = getSharedPreferences("PreferencesName", MODE_PRIVATE)
```

Тут усі дані зберігаються у вигляді XML-файлу в локальному сховищі застосунку.

Для збереження чутливих даних, таких як токени чи паролі, існує розширена версія - **EncryptedSharedPreferences**. Вона входить до складу бібліотеки Jetpack Security та забезпечує шифрування як ключів, так і значень. Це дозволяє гарантувати захист інформації, навіть якщо зловмисник отримає доступ до файлової системи пристрою. Щоб додати цю бібліотеку, потрібно підключити залежність:

```
implementation("androidx.security:security-crypto:1.1.0-alpha06")
```

Усі дані зберігатимуться у файлі /data/data/<your.package.name>/shared_prefs/secure_prefs.xml, але у зашифрованому вигляді.

Окрім налаштувань, для більш масштабного збереження даних у мобільних застосунках використовуються **бази даних**. Найпопулярнішою системою для Android є **SQLite**, яка входить у стандартний SDK. Вона дозволяє працювати з реляційними таблицями та виконувати SQL-запити. Проте робота з «чистим» SQLite часто потребує багато коду і може бути незручною.

Щоб спростити цей процес, у складі Android Jetpack була створена бібліотека **Room**. Вона надає високорівневий інтерфейс до SQLite і значно зменшує кількість рутинного коду. Основні складові роботи з Room - це **Entity**, **DAO** та **RoomDatabase**.

- **Entity** описує структуру таблиці у вигляді класу Kotlin. Наприклад:

```
@Entity(tableName = "users")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val name: String,
    val age: Int
)
```

- **DAO (Data Access Object)** містить методи для взаємодії з таблицею: зчитування, вставки, видалення та оновлення даних. Методи можуть бути suspend або повертати LiveData чи Flow:

```
@Dao
interface UserDao {
    @Query("SELECT * FROM users")
```

```
    fun getAllUsers(): LiveData<List<User>>

    @Insert
    suspend fun insert(user: User)

    @Delete
    suspend fun delete(user: User)
}
```

- **RoomDatabase** - це абстрактний клас, що забезпечує точку доступу до бази даних і конфігурацію всієї схеми:

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

Взаємодія з Room відбувається за схемою: **Repository → DAO → RoomDatabase → SQLite**. Репозиторій викликає методи DAO, які звертаються до бази даних через Room. DAO виконує SQL-запити, а результати повертаються у вигляді об'єктів Kotlin. Це дозволяє інтегрувати дані у ViewModel або напряму у UI.

Таким чином, Android надає кілька механізмів для збереження даних: від простих пар «ключ–значення» у SharedPreferences до повноцінних баз даних із Room. Вибір конкретного інструменту залежить від складності завдання: для налаштувань і невеликих параметрів достатньо SharedPreferences чи EncryptedSharedPreferences, тоді як для роботи зі складними структурами даних ефективніше використовувати Room поверх SQLite.

Завдання до лабораторної роботи:

1. Підготовка застосунку:
   а. Відкрити застосунок, над яким працювали в попередніх лабораторних роботах;
   б. Перевірити актуальність структури, UI та логіки взаємодії користувача з даними;
   в. За потреби - внести необхідні правки або оновлення.
2. Робота зі збереженням даних:
   а. Обрати відповідний інструмент для локального збереження:
      i. Для **довгострокового зберігання структурованих даних** — використати базу даних (наприклад, **Room** або альтернативну бібліотеку, з якою Ви знаєте як працювати);
      ii. Для **налаштувань користувача або токенів** — використати **SharedPreferences** або **EncryptedSharedPreferences** для захищеного зберігання.
      - Забезпечити завантаження даних з Preferences при старті застосунку і запис відповідних даних при завершенні роботи застосунку

   б. Якщо використовується **Room**:
      i. Створити сутність (entity), DAO та клас бази даних;
      ii. Реалізувати репозиторій та ViewModel для доступу до даних;
      iii. Забезпечити можливість(як мінімум) додавання, перегляду та видалення локально збережених об'єктів.
   с. Якщо використовується **будь-яка інша БД**:
      i. Забезпечити можливість(як мінімум) додавання, перегляду та видалення локально збережених об'єктів способами відповідними до обраної БД.

3. Інтеграція в інтерфейс:
   а. Відображати збережені дані у відповідних фрагментах або активності;
   б. Забезпечити зручне додавання або редагування даних через UI;
   в. Реалізувати збереження/відновлення даних при зміні орієнтації екрана.
4. Обробка виняткових ситуацій:
   а. Перевірка коректності введених даних;
   б. Обробка винятків при збереженні або отриманні даних;
   в. Створити зрозумілі для користувача повідомлення про помилки.
5. Підготовити звіт. У звіті описати:
   а. Тему застосунку та обране API;
   б. Яку базу даних використовували (Room + SQLite, інше) та яку роботу було проведено;
   в. Як використовували зберігання даних через SharedPreferences / EncryptedSharedPreferences;
   г. Скріншоти екрану з оновленим інтерфейсом та зі збереженими в БД даними.

Контрольні питання

1. Що таке WorkManager і для чого він використовується в Android-застосунках?
2. Які типи завдань можна реалізувати за допомогою WorkManager?
3. Які основні механізми збереження даних у Android-застосунках?
4. Для чого використовується Room і з яких основних компонентів вона складається (Entity, DAO, Database)?
5. Що таке SharedPreferences і які його можливості та обмеження?
6. Чим EncryptedSharedPreferences відрізняється від звичайного SharedPreferences?
7. Які дані доцільно зберігати у SharedPreferences, а які в базі даних?
8. Як забезпечити збереження даних при зміні орієнтації екрана?
9. Як організувати комбіновану роботу: Room для основних даних + SharedPreferences для налаштувань?

Рекомендована література

1. Fazio, Michael. "Kotlin and Android Development featuring Jetpack: Build Better, Safer Android Apps." (2021), 446 p.

2. Pierre-Olivier Laurence, Amanda Hinchman-Dominguez, G. Blake Meike, Mike Dunn. Programming Android with Kotlin, O'Reilly Media, Inc, 2021, 352 p.

3. NamrataBandekar, Darryl Bayliss, and Fuad Kamal. "Android Apprentice (Fourth Edition): Beginning Android Development with Kotlin", 2021, 774 p.

Додаткова література

1. Späth, Peter. Pro Android with Kotlin: developing modern mobile apps with Kotlin and Jetpack. Apress, 2022, 881 p.

2. Leeds, Dave. Kotlin: An Illustrated Guide – Softcover, TypeAlias Studios LLC, 2025, 453 p.

Навчальне видання

Методичні рекомендації до практичних занять з дисципліни «Мобільно-орієнтована розробка програмного забезпечення» для здобувачів першого (бакалаврського) рівня вищої освіти за спеціальністю 121 – Інженерія програмного забезпечення

Укладачі: Горпенко Данило Русланович
Арнаутова Олена Миколаївна