Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

ЗВІТ

з лабораторної роботи №4
з дисципліни «Сучасні мобільні операційні системи»

Тема: «Робота з мультимедійними файлами»

Варіант №4

Виконав:
студент 1 курсу, групи ІМ-51мн
Ковальов Олександр

Перевірив:
асистент, Нестерук Андрій Олександрович

Дата здачі: 25.02.2026

КИЇВ – 2026

Мета роботи. Вивчити роботу з потоками, навчитися працювати з мультимедійними файлами та з класом `AsyncTask`.

Завдання 1. Розгляньте приклад передачі даних.

```
MyAsyncTask at = new MyAsyncTask();
at.execute("url1", "url2");
doInBackground(String ... urls)
```

Завдання 2. Розгляньте приклад виведення проміжних даних.

```
@Override
protected void doInBackground (String ... urls) {
    try {
        int cnt = 0;
        for (String url: urls) {
            // обробляємо перший параметр
            ...
            // виводимо проміжні результати
            cnt++;
            publishProgress(cnt);
        }

        TimeUnit.SECONDS.sleep (1);
    } catch (InterruptedException e) {
        e.printStackTrace ();
    }
    return null;
}

@Override
protected void onProgressUpdate(Integer ... values) {
    super.onProgressUpdate (values);
    tv.setText("оброблено" + values[0] + "параметрів");
}
```

Завдання 3. Перевірте приклад створення простої асинхронної `Activity`.

```
public class MainActivity extends Activity {
    MyAsyncTask at;
    TextView tv;

    public void onCreate (Bundle savedInstance State) {
        super.onCreate(savedInstanceState);
        setContentView (R.layout.main);

        tv = (TextView) findViewById(R.id.tv);
        MyAsyncTask at = new MyAsyncTask();
        at.execute ();
    }
}
```

Завдання 4. На підставі вивчених прикладів розробити додаток, що зберігає статистику пісень, які програються на WebRadio. Для збереження пісні і назви необхідно створити базу даних, що містить таблицю з наступними полями:

1. ID

2. Виконавець
3. Назва треку
4. Час внесення запису

При запуску програми необхідно проводити перевірку підключення до Інтернету. У разі якщо сигнал мережі переривається – виводити спливаюче повідомлення (`Toast`) з попередженням про запуск в автономному режимі (доступний тільки перегляд внесених раніше записів). Після включення додаток повинен робити асинхронне опитування сервера з інтервалом 20 секунд. Якщо назва треку не збігається з останнім записом в таблиці необхідно зробити запис в базі даних. URL адреси, за якою можна отримати інформацію про поточний трек і виконавця: `https://webradio.io/api/radio/pi/current-song`

У разі успішного виконання запиту результат буде мати вигляд:

```
{ "Result": "success", "info": "Виконавець - Назва треку"}
```

У разі помилки API поверне наступний рядок:

```
{ "Result": "error", "info": "Інформація про помилку"}
```

Програма повинна дозволяти переглядати внесені в базу даних записи.

Хід роботи.

Завдання 1. У першому завданні продемонстровано механізм передачі вхідних параметрів до асинхронного завдання. Спочатку створюється екземпляр класу `MyAsyncTask`, після чого викликається метод `execute`, у який передаються необхідні аргументи, наприклад, рядки з веб-адресами. Ці передані значення автоматично потрапляють до системного методу `doInBackground`, де вони приймаються у вигляді масиву змінної довжини завдяки синтаксису `String... urls`, що дозволяє фоновому потоку отримати та почати обробку цієї інформації.

Завдання 2. Друге завдання ілюструє процес взаємодії фонового потоку з головним потоком інтерфейсу користувача для відображення проміжних результатів. У методі `doInBackground` відбувається ітерація по масиву переданих параметрів, і після обробки кожного елемента лічильник збільшується та передається у спеціальний метод `publishProgress`. Цей виклик сигналізує системі та автоматично ініціює спрацювання методу `onProgressUpdate`, який працює вже в головному потоці програми (UI Thread). Це забезпечує можливість безпечно оновити графічні компоненти, наприклад, змінити текст у `TextView` та показати поточну кількість оброблених даних без ризику блокування або збою інтерфейсу.

Завдання 3. У третьому завданні наведено базову структуру активності, яка ініціює виконання асинхронної операції відразу під час свого створення. У методі `onCreate` відбувається стандартне налаштування екрана: виклик батьківського методу, встановлення розмітки через `setContentView` та ініціалізація текстового поля за допомогою `findViewById`. Одразу після підготовки графічного інтерфейсу створюється об'єкт власного асинхронного класу `MyAsyncTask` та викликається його метод `execute` без параметрів, що запускає виконання прописаної фонової логіки паралельно з роботою активності.

Завдання 4. Для початку, так як застосунок потребує підключення до інтернету та перевірку зміни його стану, треба видати відповідні дозволи. Їх треба прописати в `AndroidManifest.xml`, перед тегом `application`:

AndroidManifest.xml

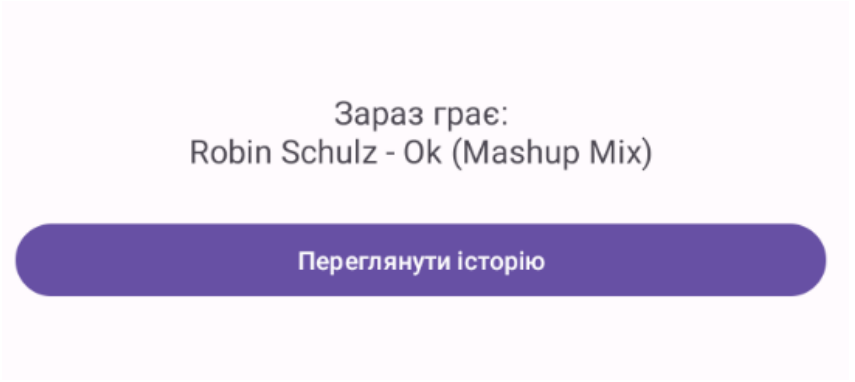
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

    <application
        android:allowBackup="true"
```

Після налаштування дозволів було розроблено клас помічника бази даних, який успадковує `SQLiteOpenHelper`. У ньому визначено структуру таблиці для збереження історії відтворення, яка складається з полів ідентифікатора, виконавця, назви треку та точного часу додавання запису. Для повноцінної роботи з цією таблицею реалізовано три основні методи. Перший відповідає за безпосереднє збереження нового треку. Другий метод виконує пошук та повертає назву останньої збереженої пісні, що критично важливо для порівняння та уникнення запису дублікатів під час постійного опитування сервера. Третій метод зчитує всю таблицю та формує з неї єдиний текстовий рядок для зручного виведення на екран історії.

Основна бізнес-логіка програми зосереджена у класі головної активності. Під час ініціалізації додаток звертається до системного сервісу `ConnectivityManager`, щоб перевірити наявність активного підключення до мережі Інтернет. Залежно від статусу з'єднання, користувач отримує спливаюче повідомлення про роботу в онлайн-режимі або попередження про перехід в автономний стан. Якщо інтернет доступний, програма запускає асинхронний процес опитування сервера. Замість застарілого класу `AsyncTask` для цього використано корутини (`Coroutines`), що дозволило створити безпечний фоновий цикл, який призупиняє своє виконання на 20 секунд між ітераціями, абсолютно

не блокуючи графічний інтерфейс користувача.

Під час кожної ітерації фоновий потік надсилає HTTP-запит до API радіостанції та отримує текстову відповідь. Далі ця відповідь обробляється за допомогою класу JSONObject, з якого вилучаються значення за ключами `artist` та `title`. Оскільки оновлення інтерфейсу дозволено лише з головного потоку, отримані дані передаються туди за допомогою контексту `Dispatchers.Main`, після чого на екрані з'являється інформація про поточну пісню. Одразу після цього назва отриманого треку порівнюється з останнім записом у локальній базі даних. Якщо виявляється розбіжність, програма генерує поточну мітку часу та ініціює збереження нової композиції.



Для реалізації вимоги щодо перегляду історії було створено окрему активність. Її графічна розмітка є максимально спрощеною і складається з єдиного текстового поля, поміщеного всередину контейнера `ScrollView`. Таке рішення гарантує можливість вільного прокручування тексту, якщо кількість збережених пісень перевищить розмір екрана. Під час відкриття цього вікна програма звертається до методу бази даних для отримання всього сформованого тексту історії та призначає його відповідному елементу інтерфейсу.

Висновок. Під час виконання лабораторної роботи було практично досліджено принципи асинхронного програмування в операційній системі Android. На основі аналізу прикладів із застарілим класом `AsyncTask` було здійснено перехід до сучасних інструментів, зокрема `Kotlin Coroutines`, що дозволило реалізувати безпечне виконання тривалих мережевих запитів у фоновому потоці без блокування інтерфейсу користувача. Було розроблено повноцінний додаток для моніторингу поточного треку радіостанції через веб-API, який успішно виконує HTTP-запити, розбирає отримані дані у форматі JSON та зберігає їхню історію у локальній базі даних SQLite з попередньою перевіркою на дублікати. Крім того, засвоєно навички роботи із системними сервісами для перевірки наявності інтернет-з'єднання та інформування користувача про поточний стан за допомогою спливаючих повідомлень `Toast`.

Контрольні запитання.

1. **Як за допомогою класу `Toast` створити спливаюче повідомлення?**
   Спливаюче повідомлення створюється шляхом виклику статичного методу `makeText` класу `Toast`, якому передаються три параметри: контекст додатку або активності, текст самого повідомлення та тривалість показу (наприклад, `Toast.LENGTH_SHORT` або `Toast.LENGTH_LONG`). Після формування об'єкта необхідно обов'язково викликати метод `show()`, щоб повідомлення з'явилося на екрані.
2. **У яких випадках необхідне логування?**
   Логування є критично необхідним на етапі розробки та дебагу додатку для відстеження потоку виконання програми, аналізу значень змінних у різних станах та виявлення прихованих помилок. Воно дозволяє розробнику фіксувати моменти виникнення виняткових ситуацій (наприклад, відсутність мережі або помилки парсингу) без переривання роботи самої програми та без виведення незрозумілої технічної інформації на екран кінцевого користувача.
3. **Що являє собою вікно `LogCat`? Які існують рівні логування?**
   Вікно `LogCat` – це вбудований інструмент середовища розробки Android Studio, який у реальному часі відображає системні повідомлення, помилки та власні текстові записи, згенеровані програмою. Існує декілька рівнів логування, які класифікують повідомлення за їхньою важливістю: `Error` (помилки, що призводять до збоїв), `Warning` (попередження про потенційні проблеми), `Info` (загальна інформаційна статистика), `Debug` (детальні дані для

дебагу) та `Verbose` (максимально детальний рівень для глибокого аналізу).

4. **Як програмно реалізувати логування?**

   Для програмної реалізації логування використовується системний клас `Log`, який надає статичні методи, що відповідають кожному рівню деталізації (наприклад, `Log.e()` для помилок, `Log.d()` для дебагу, `Log.i()` для інформації). Кожен такий метод приймає два основні рядкові параметри: тег (TAG), який зазвичай містить назву класу для зручної фільтрації у вікні `LogCat`, та безпосередньо текст самого повідомлення, яке потрібно зафіксувати.

5. **Як створити нове Activity (опишіть роботу з java-класом, layout-файлом, файлом конфігурації `AndroidManifest.xml`)?**

   Процес створення нової активності складається з трьох обов'язкових етапів. Спочатку у папці ресурсів `res/layout` створюється XML-файл розмітки, який визначає візуальну складову вікна. Потім створюється клас, який успадковується від `AppCompatActivity` чи базового класу `Activity`, де у перевизначеному методі `onCreate` відбувається прив'язка створеної розмітки за допомогою виклику `setContentView`. На фінальному етапі нову активність необхідно обов'язково зареєструвати у файлі `AndroidManifest.xml`, додавши тег `<activity>` з вказівкою імені класу всередині блоку `<application>`, інакше при спробі її відкрити програма аварійно завершить роботу.

6. **Для чого використовується контекст додатку `Context`?**

   Контекст (`Context`) є базовим інтерфейсом, який надає доступ до глобальної інформації про середовище додатка. Він використовується операційною системою як своєрідний паспорт програми, необхідний для отримання доступу до ресурсів (рядків, зображень, баз даних), ініціалізації системних сервісів (наприклад, для перевірки стану мережі), створення графічних елементів інтерфейсу, запуску нових активностей та показу спливаючих повідомлень.

Лістинг.

DatabaseHelper.kt

```
package ua.kpi.lab4

import android.content.ContentValues
import android.content.Context
import android.database.sqlite.SQLiteDatabase
import android.database.sqlite.SQLiteOpenHelper

// Helper class for managing WebRadio database
class DatabaseHelper(context: Context) :
SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION) {

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

        private const val TABLE_NAME = "songs_history"
        private const val COLUMN_ID = "id"
        private const val COLUMN_ARTIST = "artist"
        private const val COLUMN_TRACK = "track_name"
        private const val COLUMN_TIME = "time_added"
    }

    override fun onCreate(db: SQLiteDatabase) {
        // Create table for storing song statistics
        val createTableQuery = ("CREATE TABLE $TABLE_NAME ("
        + "$COLUMN_ID INTEGER PRIMARY KEY AUTOINCREMENT, "
        + "$COLUMN_ARTIST TEXT, "
        + "$COLUMN_TRACK TEXT, "
        + "$COLUMN_TIME TEXT)")
        db.execSQL(createTableQuery)
    }

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

    // Insert a new song record into the database
    fun insertSong(artist: String, track: String, time: String) {
        val db = this.writableDatabase
        val values = ContentValues()
        values.put(COLUMN_ARTIST, artist)
        values.put(COLUMN_TRACK, track)
        values.put(COLUMN_TIME, time)
        db.insert(TABLE_NAME, null, values)
        db.close()
    }

    // Retrieve the name of the last saved track to prevent duplicates
    fun getLastTrackName(): String? {
        val db = this.readableDatabase
        var lastTrack: String? = null
        val query = "SELECT $COLUMN_TRACK FROM $TABLE_NAME ORDER BY $COLUMN_ID DESC LIMIT 1"
        val cursor = db.rawQuery(query, null)

        if (cursor.moveToFirst()) {
            lastTrack = cursor.getString(0)
        }
```

```
        cursor.close()
        db.close()
        return lastTrack
    }

    // Retrieve all saved songs as a formatted string
    fun getAllSongs(): String {
        val db = this.readableDatabase
        val cursor = db.rawQuery("SELECT * FROM $TABLE_NAME ORDER BY $COLUMN_ID DESC", null)
        val stringBuilder = StringBuilder()

        if (cursor.moveToFirst()) {
            do {
                val artist = cursor.getString(cursor.getColumnIndexOrThrow(COLUMN_ARTIST))
                val track = cursor.getString(cursor.getColumnIndexOrThrow(COLUMN_TRACK))
                val time = cursor.getString(cursor.getColumnIndexOrThrow(COLUMN_TIME))
                stringBuilder.append("Час: $time\nВиконавець: $artist\nТрек: $track\n\n")
            } while (cursor.moveToNext())
        } else {
            stringBuilder.append("Історія порожня.")
        }
        cursor.close()
        db.close()
        return stringBuilder.toString()
    }
}
```

MainActivity.kt

```
package ua.kpi.lab4

import android.content.Intent
import android.net.ConnectivityManager
import android.net.NetworkCapabilities
import android.os.Bundle
import android.widget.Button
import android.widget.TextView
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import androidx.lifecycle.lifecycleScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.delay
import kotlinx.coroutines.isActive
import kotlinx.coroutines.launch
import kotlinx.coroutines.withContext
import org.json.JSONObject
import java.io.BufferedReader
import java.io.InputStreamReader
import java.net.HttpURLConnection
import java.net.URL
import java.text.SimpleDateFormat
import java.util.Date
import java.util.Locale

class MainActivity : AppCompatActivity() {

    private lateinit var dbHelper: DatabaseHelper
    private lateinit var tvCurrentStatus: TextView

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

```
        dbHelper = DatabaseHelper(this)
        tvCurrentStatus = findViewById(R.id.tvCurrentStatus)
        val btnViewHistory = findViewById<Button>(R.id.btnViewHistory)

        btnViewHistory.setOnClickListener {
            startActivity(Intent(this, HistoryActivity::class.java))
        }

        // Check internet connection on startup
        if (isNetworkAvailable()) {
            Toast.makeText(this, "Онлайн режим. Опитування сервера...", Toast.LENGTH_SHORT).show()
            startPollingServer()
        } else {
            Toast.makeText(
            this,
            "Автономний режим. Доступний лише перегляд бази.",
            Toast.LENGTH_LONG
            ).show()
            tvCurrentStatus.text = "Немає підключення до мережі"
        }
    }

    // Helper method to check internet connectivity
    private fun isNetworkAvailable(): Boolean {
        val connectivityManager = getSystemService(CONNECTIVITY_SERVICE) as ConnectivityManager
        val network = connectivityManager.activeNetwork ?: return false
        val activeNetwork = connectivityManager.getNetworkCapabilities(network) ?: return false
        return activeNetwork.hasCapability(NetworkCapabilities.NET_CAPABILITY_INTERNET)
    }

    // Coroutine to poll the server every 20 seconds
    private fun startPollingServer() {
        lifecycleScope.launch(Dispatchers.IO) {
            while (isActive) {
                if (isNetworkAvailable()) {
                    fetchCurrentSong()
                }
                // Delay for 20 seconds
                delay(20000)
            }
        }
    }

    // Fetch data from API and process it
    private suspend fun fetchCurrentSong() {
        try {
            val url = URL("https://webradio.io/api/radio/pi/current-song")
            val connection = url.openConnection() as HttpURLConnection
            connection.requestMethod = "GET"
            connection.connectTimeout = 5000

            if (connection.responseCode == HttpURLConnection.HTTP_OK) {
                val reader = BufferedReader(InputStreamReader(connection.inputStream))
                val response = StringBuilder()
                var line: String?
                while (reader.readLine().also { line = it } != null) {
                    response.append(line)
                }
                reader.close()

                parseAndSaveData(response.toString())
            }
```

```
            connection.disconnect()
        } catch (e: Exception) {
            e.printStackTrace()
            updateUIStatus("Помилка з'єднання з сервером")
        }
    }

    // Parse JSON and interact with database
    private suspend fun parseAndSaveData(jsonString: String) {
        try {
            val jsonObject = JSONObject(jsonString)

            // Fetching data
            val artist = jsonObject.optString("artist", "Невідомий виконавець")
            val track = jsonObject.optString("title", "Невідомий трек")

            updateUIStatus("Зараз грає:\n$artist - $track")

            val lastTrackInDb = dbHelper.getLastTrackName()
            if (lastTrackInDb != track) {
                val currentTime =
                SimpleDateFormat("yyyy-MM-dd HH:mm:ss", Locale.getDefault()).format(Date())
                dbHelper.insertSong(artist, track, currentTime)
            }

        } catch (e: Exception) {
            e.printStackTrace()
            updateUIStatus("Помилка парсингу: ${e.message}\nСирі дані:\n$jsonString")
        }
    }

    // Helper to update TextView from background threads
    private suspend fun updateUIStatus(message: String) {
        withContext(Dispatchers.Main) {
            tvCurrentStatus.text = message
        }
    }
}
```

activity_main.xml

```
<?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:layout_width="match_parent"
android:layout_height="match_parent"
android:padding="16dp">

<TextView
android:id="@+id/tvCurrentStatus"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="Ініціалізація..."
android:textAlignment="center"
android:textSize="18sp"
app:layout_constraintBottom_toTopOf="@+id/btnViewHistory"
app:layout_constraintTop_toTopOf="parent"
app:layout_constraintVertical_chainStyle="packed" />

<Button
android:id="@+id/btnViewHistory"
```

```
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_marginTop="24dp"
android:text="Переглянути історію"
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintTop_toBottomOf="@+id/tvCurrentStatus" />

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

HistoryActivity.kt

```
package ua.kpi.lab4

import android.os.Bundle
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity

class HistoryActivity : AppCompatActivity() {
    private lateinit var dbHelper: DatabaseHelper

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

        dbHelper = DatabaseHelper(this)

        val tvDatabaseContent = findViewById<TextView>(R.id.tvDatabaseContent)

        // Fetch and display all data from the database
        val allData = dbHelper.getAllSongs()
        tvDatabaseContent.text = allData
    }
}
```

activity_history.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:padding="16dp">

<TextView
android:id="@+id/tvDatabaseContent"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:textColor="#000000"
android:textSize="18sp" />

</ScrollView>
```