MOBILE APPLICATION DEVELOPMENT USING ANDROID

MODULE 3

Activities, Intents & Data Persistence

Comprehensive Study Material

TABLE OF CONTENTS

1. INTENTS OVERVIEW

1.1 What is an Intent?

An Intent is a messaging object used to request an action from another app component. It serves three primary purposes:

- Start an Activity: Launch a new screen or activity
- Start a Service: Initiate background operations
- Broadcast a Message: Send system-wide announcements

1.2 Intent Types

Type	Description	Use Case
Explicit Intent	Specifies the exact component to start by name	Launching a specific activity within your app
Implicit Intent	Declares a general action to perform; system resolves appropriate component	Opening a URL, sharing content, or taking a photo

1.3 Explicit Intent Example

```
// Method 1: Direct class reference
Intent intent = new Intent(this, SecondActivity.class);
startActivity(intent);

// Method 2: Using context and class name
Intent intent = new Intent(getApplicationContext(), SecondActivity.class);
startActivity(intent);

// Kotlin version
val intent = Intent(this, SecondActivity::class.java)
startActivity(intent)
```

1.4 Implicit Intent Examples

```
// Open a URL
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("https://developer.android.com"));
startActivity(intent);

// Make a phone call
Intent intent = new Intent(Intent.ACTION_DIAL);
intent.setData(Uri.parse("tel:+1234567890"));
startActivity(intent);

// Share content
Intent intent = new Intent(Intent.ACTION_SEND);
intent.setType("text/plain");
intent.putExtra(Intent.EXTRA_TEXT, "Check out this app!");
startActivity(Intent.createChooser(intent, "Share via"));
```

1.5 Intent Resolution for Implicit Intents

When you use an implicit intent, the Android system performs three steps:

- Intent Filter Matching: Compares the intent against all installed apps' intent filters
- Action Matching: Checks if any app handles the specified action
- Data/Type Matching: Verifies data type compatibility

2. ACTIVITY COMMUNICATION: SENDING DATA

2.1 Intent Extras — Primitive Data Types

```
// === Sending Activity (MainActivity) ===
btnNavigate.setOnClickListener(v -> {
    Intent intent = new Intent(MainActivity.this, SecondActivity.class);
    intent.putExtra("USERNAME", etUsername.getText().toString());
    intent.putExtra("EMAIL",    etEmail.getText().toString());
    intent.putExtra("SUBSCRIBE", cbSubscribe.isChecked());
    intent.putExtra("AGE",      25);
    intent.putExtra("SALARY",   50000.0);
    startActivity(intent);
});

// === Receiving Activity (SecondActivity) ===
Intent intent   = getIntent();
String username = intent.getStringExtra("USERNAME");
String email    = intent.getStringExtra("EMAIL");
boolean subscribed = intent.getBooleanExtra("SUBSCRIBE", false);
int     age     = intent.getIntExtra("AGE", 0);
double  salary  = intent.getDoubleExtra("SALARY", 0.0);
```

2.2 Bundle for Organized Data

```
// Pack data into Bundle
Bundle dataBundle = new Bundle();
dataBundle.putString("USERNAME",  etUsername.getText().toString());
dataBundle.putString("EMAIL",     etEmail.getText().toString());
dataBundle.putBoolean("SUBSCRIBE", cbSubscribe.isChecked());
dataBundle.putInt("AGE", 25);

Intent intent = new Intent(MainActivity.this, SecondActivity.class);
intent.putExtras(dataBundle);
startActivity(intent);

// Unpack in receiving activity
Bundle bundle = getIntent().getExtras();
if (bundle != null) {
    String username = bundle.getString("USERNAME");
    int    age      = bundle.getInt("AGE", 0);
}
```

3. PASSING COMPLEX OBJECTS

3.1 Serializable Interface

Serializable is simpler to implement but slower — uses Java reflection under the hood.

```
public class User implements Serializable {
    private String name, email;
    private int age;
    private boolean subscribed;
    // constructor, getters ...
}

// Sending
User user = new User("John Doe", "john@example.com", 25, true);
intent.putExtra("USER_OBJECT", user);

// Receiving
User received = (User) getIntent().getSerializableExtra("USER_OBJECT");
```

3.2 Parcelable Interface (Recommended)

Parcelable is more efficient than Serializable — no reflection, explicit marshalling.

```
public class Student implements Parcelable {
    private String name, email;
    private int age;
    private List<String> courses;

    protected Student(Parcel in) {
        name    = in.readString();
        email   = in.readString();
        age     = in.readInt();
        courses = in.createStringArrayList();
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(name);
        dest.writeString(email);
        dest.writeInt(age);
        dest.writeStringList(courses);
    }

    public static final Creator<Student> CREATOR = new Creator<Student>() {
        @Override
        public Student createFromParcel(Parcel in) { return new Student(in); }
        @Override
        public Student[] newArray(int size) { return new Student[size]; }
    };
    @Override public int describeContents() { return 0; }
}
```

3.3 Kotlin @Parcelize (Shorthand)

```
// Requires 'kotlin-parcelize' plugin in build.gradle
```

```
@Parcelize
data class Product(
    val id: Int,
    val name: String,
    val price: Double,
    val description: String,
    val imageUrl: String? = null
) : Parcelable

// Sending
val product = Product(1, "Smartphone", 699.99, "Latest 5G model")
intent.putExtra("PRODUCT", product)

// Receiving
val product = intent.getParcelableExtra<Product>("PRODUCT")
```

4. START ACTIVITY FOR RESULT

4.1 Legacy Approach

```
// Sending Activity
private static final int REQUEST_CODE_SELECT = 1;

btnSelect.setOnClickListener(v -> {
    Intent intent = new Intent(this, SelectActivity.class);
    startActivityForResult(intent, REQUEST_CODE_SELECT);
});

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == REQUEST_CODE_SELECT && resultCode == RESULT_OK) {
        String selected = data.getStringExtra("SELECTED_ITEM");
        tvResult.setText("Selected: " + selected);
    }
}

// Receiving Activity — send result back
Intent resultIntent = new Intent();
resultIntent.putExtra("SELECTED_ITEM", selectedValue);
setResult(RESULT_OK, resultIntent);
finish();
```

4.2 ActivityResultLauncher (Modern / AndroidX)

> *The ActivityResultLauncher API replaces the deprecated startActivityForResult pattern.*

```
private final ActivityResultLauncher<Intent> selectLauncher =
    registerForActivityResult(
        new ActivityResultContracts.StartActivityForResult(),
        result -> {
            if (result.getResultCode() == RESULT_OK) {
                Intent data = result.getData();
                if (data != null) {
                    String selected = data.getStringExtra("SELECTED_ITEM");
                    tvResult.setText("Selected: " + selected);
                }
            }
        }
    );

btnSelect.setOnClickListener(v -> {
    selectLauncher.launch(new Intent(this, SelectActivity.class));
});
```

5. SHARED PREFERENCES

5.1 Overview

SharedPreferences provides a lightweight mechanism for storing simple key-value pairs of primitive data types.

- Stores: boolean, float, int, long, String
- Private to the app — not accessible by other apps
- Persistent across app restarts
- Ideal for user preferences and settings

5.2 Getting SharedPreferences

```
// Method 1: Default preferences
SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(this);

// Method 2: Named file (preferred – explicit naming)
SharedPreferences prefs = getSharedPreferences("MyPrefs", Context.MODE_PRIVATE);

// Method 3: Activity-specific
SharedPreferences prefs = getPreferences(Context.MODE_PRIVATE);
```

5.3 Reading & Writing

```
SharedPreferences prefs = getSharedPreferences("MyPrefs", Context.MODE_PRIVATE);
SharedPreferences.Editor editor = prefs.edit();

// Write
editor.putString("USERNAME",    "JohnDoe");
editor.putInt("AGE",            25);
editor.putBoolean("IS_LOGGED_IN", true);
editor.putFloat("RATING",       4.5f);
editor.apply();       // async – recommended
// editor.commit();   // sync  – returns boolean

// Read (second arg = default value)
String  username  = prefs.getString("USERNAME",     "Guest");
int     age       = prefs.getInt("AGE",            0);
boolean loggedIn  = prefs.getBoolean("IS_LOGGED_IN", false);

// Remove / clear
editor.remove("AGE");
editor.clear();
editor.apply();
```

5.4 apply() vs commit()

Method	Execution	Return Value	Recommended

apply()	Asynchronous	void	Yes — avoids blocking UI thread
commit()	Synchronous	boolean (success)	Only if you need immediate confirmation

6. SQLITE DATABASE

6.1 Overview

SQLite is a lightweight, embedded SQL database that stores data in a structured format.

- Full SQL support
- ACID transactions
- Stores data in a single file on disk
- Accessible only to your app — no server required

6.2 Database Helper Class

```
public class DatabaseHelper extends SQLiteOpenHelper {

    private static final String DATABASE_NAME    = "StudentDB.db";
    private static final int    DATABASE_VERSION = 1;

    public static final String TABLE_STUDENTS = "students";
    public static final String COLUMN_ID      = "_id";
    public static final String COLUMN_NAME    = "name";
    public static final String COLUMN_EMAIL   = "email";
    public static final String COLUMN_AGE     = "age";
    public static final String COLUMN_MARKS   = "marks";

    private static final String CREATE_TABLE =
        "CREATE TABLE " + TABLE_STUDENTS + " (" +
        COLUMN_ID    + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
        COLUMN_NAME  + " TEXT NOT NULL, " +
        COLUMN_EMAIL + " TEXT UNIQUE, " +
        COLUMN_AGE   + " INTEGER, " +
        COLUMN_MARKS + " REAL" +
        ")";

    public DatabaseHelper(Context ctx) {
        super(ctx, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db)    { db.execSQL(CREATE_TABLE); }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldV, int newV) {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_STUDENTS);
        onCreate(db);
    }
}
```

6.3 CRUD Operations

```
// CREATE
public long insert(String name, String email, int age, double marks) {
    ContentValues v = new ContentValues();
    v.put(COLUMN_NAME, name);  v.put(COLUMN_EMAIL, email);
    v.put(COLUMN_AGE, age);    v.put(COLUMN_MARKS, marks);
    return db.insert(TABLE_STUDENTS, null, v);
```

```
}

// READ ALL
public List<Student> getAll() {
    List<Student> list = new ArrayList<>();
    Cursor c = db.query(TABLE_STUDENTS, null, null, null, null, null,
                        COLUMN_NAME + " ASC");
    while (c.moveToNext()) { list.add(cursorToStudent(c)); }
    c.close();  return list;
}

// UPDATE
public int update(Student s) {
    ContentValues v = new ContentValues();
    v.put(COLUMN_NAME, s.getName());
    v.put(COLUMN_MARKS, s.getMarks());
    return db.update(TABLE_STUDENTS, v,
                     COLUMN_ID + " = ?", new String[]{ String.valueOf(s.getId()) });
}

// DELETE
public boolean delete(long id) {
    return db.delete(TABLE_STUDENTS,
                     COLUMN_ID + " = ?", new String[]{ String.valueOf(id) }) > 0;
}
```

7. ROOM PERSISTENCE LIBRARY

7.1 Overview

Room is an ORM (Object-Relational Mapping) library that provides an abstraction layer over SQLite.

- Compile-time SQL query verification
- Minimal boilerplate — annotations do the heavy lifting
- Full type safety
- Native LiveData and RxJava integration
- Supports database migrations

7.2 Three Main Components

Component	Role	Annotation
Entity	Represents a database table (a data model class)	@Entity
DAO	Data Access Object — interface declaring database operations	@Dao
Database	Main entry point; ties entities and DAOs together	@Database

7.3 Gradle Dependency

```
// build.gradle (app)
def room_version = "2.6.1"
implementation "androidx.room:room-runtime:$room_version"
annotationProcessor "androidx.room:room-compiler:$room_version"

// Kotlin
kapt "androidx.room:room-compiler:$room_version"
implementation "androidx.room:room-ktx:$room_version"
```

7.4 Entity Definition

```
@Entity(tableName = "students")
public class Student {

    @PrimaryKey(autoGenerate = true)
    private int id;

    @ColumnInfo(name = "full_name")
    private String name;

    @ColumnInfo(name = "email_address")
    private String email;

    private int    age;
    private String department;
```

```
    private double marks;

    // Kotlin shorthand
    // @Parcelize
    // data class Student(val id: Int, val name: String, ...) : Parcelable
}
```

7.5 DAO Interface

```
@Dao
public interface StudentDao {

    @Insert
    long insertStudent(Student student);

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void insertOrReplace(Student student);

    @Query("SELECT * FROM students ORDER BY full_name ASC")
    List<Student> getAllStudents();

    @Query("SELECT * FROM students WHERE id = :studentId")
    Student getStudentById(int studentId);

    @Query("SELECT * FROM students WHERE full_name LIKE :term")
    List<Student> searchStudents(String term);

    @Query("SELECT * FROM students WHERE marks >= :min ORDER BY marks DESC")
    List<Student> getTopStudents(double min);

    // LiveData variant — auto-updates observers on changes
    @Query("SELECT * FROM students")
    LiveData<List<Student>> getAllStudentsLive();

    @Update
    int updateStudent(Student student);

    @Delete
    int deleteStudent(Student student);

    @Query("DELETE FROM students")
    void deleteAll();
}
```

7.6 Database Class

```
@Database(entities = {Student.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {

    private static AppDatabase instance;

    public abstract StudentDao studentDao();

    public static synchronized AppDatabase getInstance(Context ctx) {
        if (instance == null) {
            instance = Room.databaseBuilder(
```

```
                ctx.getApplicationContext(),
                AppDatabase.class,
                "student_database"
            ).build();
        }
        return instance;
    }
}
```

8. DATA PERSISTENCE — COMPARISON

Feature	SharedPreferences	SQLite (Raw)	Room
Data Type	Primitive key-value pairs	Relational tables	Relational tables
Complexity	Simple	High (boilerplate)	Medium
SQL Support	None	Full SQL	Annotation-based SQL
Type Safety	No	No	Yes (compile-time)
Compile-time Check	No	No	Yes
LiveData Support	No	No	Yes
Best For	Settings & preferences	Complex data (legacy)	Complex data (modern)

9. PRACTICE EXERCISES

Exercise 1 — User Login with SharedPreferences

- Create a login screen with username and password fields
- Save login status in SharedPreferences on successful login
- Show a Welcome message if the user is already logged in at app start
- Implement logout that clears the stored credentials

Exercise 2 — Student Database with SQLite

- Create a Student table: id, name, email, age, department, marks
- Implement all CRUD operations
- Display students in a ListView
- Add search by name functionality
- Allow deletion via long-press with confirmation dialog

Exercise 3 — User Profile with Room

- Create a User entity: id, name, email, phone
- Implement DAO with insert, update, delete, getAll
- Display and allow editing of user profile

Exercise 4 — Data Transfer Between Activities

- Create a form with multiple input fields
- Pass primitive types via Intent extras
- Pass a custom object using Parcelable
- Pass grouped data using Bundle

Exercise 5 — Settings Screen

- Create a settings screen: theme, notifications toggle, language
- Persist all settings using SharedPreferences
- Apply settings when app restarts

10. KEY CONCEPTS SUMMARY

Concept	Purpose	Key Methods / Annotations
Explicit Intent	Launch a specific component	new Intent(context, TargetClass.class)
Implicit Intent	Request a generic action	new Intent(Intent.ACTION_VIEW)
Intent Extras	Pass primitive data via intent	putExtra() / getStringExtra()
Bundle	Group multiple extras together	putExtras() / getExtras()
Serializable	Pass objects between activities	putSerializable() / getSerializableExtra()
Parcelable	Efficient object passing	writeToParcel() / CREATOR
startActivityForResult	Get a result from an activity	onActivityResult() / setResult()
ActivityResultLauncher	Modern result-callback API	registerForActivityResult()
SharedPreferences	Persistent key-value storage	getSharedPreferences() / edit() / apply()
SQLite	Raw relational database	SQLiteOpenHelper / ContentValues / Cursor
Room — @Entity	Declares a table	@Entity, @PrimaryKey, @ColumnInfo
Room — @Dao	Database operations interface	@Insert, @Update, @Delete, @Query
Room — @Database	App database entry point	@Database(entities, version)

11. INTERVIEW QUESTIONS

1. What is an Intent and what are its two types?
2. Explain the difference between explicit and implicit intents with examples.
3. How do you pass data between activities? List at least three methods.
4. What is the difference between Serializable and Parcelable? When should you use each?
5. Explain the purpose and limitations of SharedPreferences.
6. How do you create and manage a SQLite database in Android?
7. What advantages does Room provide over raw SQLite?
8. Explain the three main components of Room — Entity, DAO, and Database.
9. What is the difference between apply() and commit() in SharedPreferences?
10. How do you handle database schema migrations in Room?

12. REFERENCES

- Android Developers Documentation — Intents and Intent Filters
- Android Developers Documentation — Save key-value data (SharedPreferences)
- Android Developers Documentation — Save data in a local database using Room
- Android Developers Documentation — Save data using SQLite

End of Module 3 Study Material