

안드로이드 프로그래밍

3장. 프래그먼트

Introduction to Fragments

Fragment Definition and Purpose

- A fragment is a modular section of an activity that:
 - Can exist in multiple activities
 - Allows multiple fragments within one activity
 - Encapsulates UI and functionality
 - Promotes code reusability across different screen sizes

Benefits for Multi-Screen Development

- Dual-pane layouts for tablets:
 - Phone: Display list only due to space constraints
 - Tablet: Display list + detailed content simultaneously
 - Same fragment reused across different form factors
 - Eliminates code duplication and simplifies maintenance

The Fragment Lifecycle

Fragments have their own lifecycle with callbacks at specific stages of creation, running state, and destruction, closely tied to the host activity's lifecycle.

Fragment Lifecycle Overview

- Key characteristics:
 - Similar to activity lifecycle but interleaved with activity callbacks
 - Fragment callbacks occur before activity callbacks during creation
 - Fragments must extend the parent Fragment class
 - Lifecycle tied to containing activity's lifecycle

Creation Phase Callbacks

- `onAttach(context: Context)`
 - Fragment becomes linked to activity
 - Allows activity reference (neither fully created yet)
- `onCreate(savedInstanceState: Bundle?)`
 - Fragment initialization phase
 - No UI available yet (`no setContentView`)
 - Use `savedInstanceState` for state restoration

View Creation Callbacks

- `onCreateView(inflater, container, savedInstanceState): View?`
 - Creates fragment layout - returns View instead of setting it
 - Views available but prefer manipulation in `onViewCreated`
- `onViewCreated(view: View, savedInstanceState: Bundle?)`
 - Setup views and interactivity (onClickListeners)
 - Called between creation and visibility
 - Preferred location for view manipulation

Activity Integration Callbacks

- `onActivityCreated(context: Context)`
 - Called after activity's `onCreate` completes
 - Most view state initialization done
 - Place for final setup if required
- `onStart()`
 - Fragment about to become visible
- `onResume()`
 - Fragment ready for user interaction

Destruction Phase Callbacks

- `onPause()` - App going to background or partially covered
- `onStop()` - Fragment no longer visible
- `onDestroyView()` - Final cleanup before destruction, layout view removed
- `onDestroy()` - Fragment being destroyed (app killed or replaced)
- `onDetach()` - Fragment detached from activity

Exercise 3.01: Basic Fragment Implementation

- Objective: Create and add basic fragment to examine lifecycle callbacks
- Key steps:
 - Create FragmentLifecycle app with MainFragment
 - Add fragment to activity XML with <fragment> element
 - Implement lifecycle callbacks with logging
 - Observe interleaved callback order between fragment and activity

Exercise 3.01: Basic Fragment Implementation

- Fragment XML Integration

```
<fragment
    android:id="@+id/main_fragment"
    android:name="com.example.fragmentlifecycle.MainFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Exercise 3.01: Basic Fragment Implementation

- Lifecycle Callback Sequence
 - App startup order:
 - Fragment: onAttach → onCreate → onCreateView → onViewCreated
 - Activity: onCreate
 - Fragment: onActivityCreated → onStart → onResume
 - Activity: onStart → onResume
 - App destruction: Fragment callbacks first (inward to outward), then activity

Exercise 3.02: Multiple Static Fragments

- Objective: Add two fragments with separate UI and functionality
- Components created:
 - CounterFragment: Increment/decrement counter with +/- buttons
 - ColorFragment: Change text color with Red/Green/Blue buttons
 - Both fragments in single activity using LinearLayout with weights

Exercise 3.02: Multiple Static Fragments

- CounterFragment Implementation

- Features:

- var counter = 0 property for state
 - Plus button: counter++ and update display
 - Minus button: counter-- (only if > 0)
 - Uses onViewCreated for button setup and findViewById for view references

Exercise 3.02: Multiple Static Fragments

- Layout Weight Distribution

```
<fragment  
    android:layout_height="0dp"  
    android:layout_weight="2"/>  <!-- CounterFragment -->  
<fragment  
    android:layout_height="0dp"  
    android:layout_weight="1"/>  <!-- ColorFragment -->
```

- Result: CounterFragment occupies 2/3 height, ColorFragment 1/3 height

Exercise 3.02: Multiple Static Fragments

- ColorFragment Functionality
 - Implementation:

```
redButton.setOnClickListener {  
    helloWorldTextView.setTextColor(Color.RED)  
}
```

- Demonstrates: Fragment encapsulation - self-contained UI logic and features, promoting reusability across multiple activities

Static Fragments and Dual-Pane Layouts

Static fragments defined in activity XML layouts, with different resource configurations for phones vs tablets using screen size qualifiers.

Screen Size Qualifiers

- `sw600dp` qualifier:
 - `sw` = shortest width
 - `600dp` = minimum width for 7" tablets and larger
 - Creates `res/layout-sw600dp/` folder for tablet-specific layouts
 - Enables dual-pane layouts when screen space permits

Dual-Pane Layout Concept

- Pane definition: Self-contained part of user interface
- Implementation strategy:
 - Phone: Single pane - list → separate screen for details
 - Tablet: Dual pane - list + details simultaneously
 - Same fragments reused across form factors
 - One pane can interact with another to update content

Exercise 3.03: Star Signs Dual-Pane App

- Objective: Create app displaying star signs list with details, using different layouts for phones and tablets
- Components:
 - ListFragment: Scrollable list of 12 star signs
 - DetailFragment: Display symbol, date range for selected sign
 - StarSignListener interface for fragment-activity communication

Exercise 3.03: Star Signs Dual-Pane App

- ListFragment Implementation
 - Layout structure:

```
<ScrollView>
    <LinearLayout android:orientation="vertical">
        <TextView android:text="@string/star_signs"/>
        <TextView android:id="@+id/aquarius"/>
        <!-- 11 more star signs -->
    </LinearLayout>
</ScrollView>
```

- ScrollView prevents content cutoff when exceeding screen height

Exercise 3.03: Star Signs Dual-Pane App

- Fragment Communication Pattern
 - StarSignListener Interface:

```
interface StarSignListener {  
    fun onSelected(id: Int)  
}
```

- Implementation:
 - Fragment checks if activity implements interface in onAttach
 - Fragment calls starSignListener.onSelected(starSign.id) on click
 - Activity receives callback and handles navigation logic

Exercise 3.03: Star Signs Dual-Pane App

- DetailFragment Data Binding
 - setStarSignData() function:

```
when (starSignId) {  
    R.id.aquarius -> {  
        starSign?.text = getString(R.string.aquarius)  
        symbol?.text = getString(R.string.symbol, "Water Carrier")  
        dateRange?.text = getString(R.string.date_range, "January 20 - February 18")  
    }  
}
```

- String formatting: %s placeholder replaced with passed values

Exercise 3.03: Star Signs Dual-Pane App

- Responsive Layout Implementation
 - Phone layout (`activity_main.xml`):
 - Single ListFragment occupying full screen
 - Selection opens DetailActivity with DetailFragment
 - Tablet layout (`activity_main.xml` in `layout-sw600dp`):
 - LinearLayout with horizontal orientation
 - ListFragment (`weight=1`) + divider + DetailFragment (`weight=2`)
 - 1:2 width ratio for optimal tablet display

Exercise 3.03: Star Signs Dual-Pane App

- Dynamic Layout Detection

- isDualPane detection:

```
isDualPane = findViewById<View>(R.id.star_sign_detail) != null
```

- Navigation logic:

- If dual-pane: Update DetailFragment directly via supportFragmentManager
 - If single-pane: Launch DetailActivity with intent extras
 - Same fragments work in both configurations

Dynamic Fragments

Fragments added, replaced, and removed at runtime using ViewGroup containers, providing greater flexibility than static XML-defined fragments.

Dynamic vs Static Fragments

- Static fragments:
 - Defined in XML at compile time
 - Always inflated in layouts
 - Better for fixed UI interactions
 - Example: List → detail selection
- Dynamic fragments:
 - Added at runtime responding to user actions
 - More flexible - can be removed when not needed
 - Better for complex user journeys (3-4+ fragments)
 - Requires ViewGroup container

Exercise 3.04: Dynamic Star Signs App

- Objective: Rebuild star signs app using dynamic fragment management
- Key changes:
 - FragmentContainerView as container instead of static XML fragments
 - Fragment transactions for add/replace operations
 - Factory method pattern for passing arguments
 - Back stack management

Exercise 3.04: Dynamic Star Signs App

- FragmentContainerView Setup

```
<androidx.fragment.app.FragmentContainerView  
    android:id="@+id/fragment_container"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent" />
```

- Dependency required:

```
implementation 'androidx.fragment:fragment-ktx:1.5.6'
```

Exercise 3.04: Dynamic Star Signs App

- Fragment Transaction Pattern
 - Adding initial fragment:

```
val listFragment = ListFragment()
supportFragmentManager.beginTransaction()
    .add(frameLayout.id, listFragment)
    .commit()
```

- Replacing with detail:

```
supportFragmentManager.beginTransaction()
    .replace(frameLayout.id, detailFragment)
    .addToBackStack(null)
    .commit()
```

Exercise 3.04: Dynamic Star Signs App

- Fragment Factory Method Pattern
 - DetailFragment companion object:

```
companion object {
    private const val STAR_SIGN_ID = "STAR_SIGN_ID"

    fun newInstance(starSignId: Int) = DetailFragment().apply {
        arguments = Bundle().apply {
           .putInt(STAR_SIGN_ID, starSignId)
        }
    }
}
```

- Benefits: Type-safe argument passing, consistent fragment creation

Exercise 3.04: Dynamic Star Signs App

- Argument Retrieval and Back Stack
 - In onViewCreated:

```
val starSignId = arguments?.getInt(STAR_SIGN_ID, 0) ?: 0
setStarSignData(starSignId)
```

- Back stack management:
 - .addToBackStack(null) prevents app exit on back press
 - Returns to ListFragment instead of closing app
 - Manages fragment navigation history automatically

Jetpack Navigation

Android Jetpack Navigation component reduces boilerplate code and simplifies fragment management through navigation graphs and destinations.

Navigation Component Benefits

- Problems solved:
 - Boilerplate code reduction in fragment management
 - Complex back stack management
 - Multiple fragment add/remove/replace operations
 - User journey organization and documentation
- Solution: Navigation graphs with destinations and actions

Exercise 3.05: Jetpack Navigation Implementation

- Objective: Rebuild star signs app using Navigation component
- Dependencies required:

```
implementation "androidx.navigation:navigation-fragment-ktx:2.5.3"
implementation "androidx.navigation:navigation-ui-ktx:2.5.3"
```

- Components: Navigation graph, NavHostFragment, navigation actions

Exercise 3.05: Jetpack Navigation Implementation

- Navigation Graph Structure

```
<navigation android:id="@+id/nav_graph"
    app:startDestination="@+id/starSignList">

    <fragment android:id="@+id/starSignList"
        android:name="com.example.jetpackfragments.ListFragment">
        <action android:id="@+id/star_sign_id_action"
            app:destination="@+id/starSign" />
    </fragment>

    <fragment android:id="@+id/starSign"
        android:name="com.example.jetpackfragments.DetailFragment" />
</navigation>
```

Exercise 3.05: Jetpack Navigation Implementation

- Navigation Graph Concepts
 - Destinations: Individual fragments or activities in navigation flow
 - Actions: Links between destinations with unique IDs
 - Start Destination: Entry point defined by `app:startDestination`
- Benefits:
 - Visual representation of app navigation
 - Centralized navigation logic
 - Type-safe argument passing

Exercise 3.05: Jetpack Navigation Implementation

- NavHostFragment Configuration

```
<androidx.fragment.app.FragmentContainerView  
    android:id="@+id/nav_host_fragment"  
  
    android:name="androidx.navigation.fragment.NavHostFragment"  
    app:defaultNavHost="true"  
    app:navGraph="@navigation/nav_graph" />
```

- Key attributes:
 - app:navGraph: Links to navigation graph
 - app:defaultNavHost: Controls back navigation
 - Can have multiple NavHostFragments for dual-pane layouts

Exercise 3.05: Jetpack Navigation Implementation

- Simplified Navigation Implementation
 - Fragment navigation with arguments:

```
val fragmentBundle = Bundle()
fragmentBundle.putInt(STAR_SIGN_ID, starSign.id)

starSign.setOnClickListener(
    Navigation.createNavigateOnClickListener(
        R.id.star_sign_id_action, fragmentBundle
    )
)
```

- Result: Significant boilerplate reduction, automatic lifecycle management

Activity 3.01: Planet Quiz Application

Practical application creating a quiz about Solar System planets using fragment concepts learned throughout the chapter.

Quiz Requirements

- Three questions:
 - "What is the largest planet?"
 - "Which planet has the most moons?"
 - "Which planet spins on its side?"
- Answer options: All 8 planets (Mercury through Neptune)
- Correct answers with explanations:
 - Jupiter: Largest planet, 2.5x mass of all others combined
 - Saturn: Most moons with 82 total
 - Uranus: Spins on its side, axis nearly perpendicular to Sun

Implementation Approach Options

- Fragment architecture choices:
 - Static fragments: Questions + answers in XML
 - Dynamic fragments: Runtime fragment management
 - Jetpack Navigation: Navigation graph approach
 - Custom combination: Mixed approach
- Recommended structure: 2+ fragments separating UI and logic components

Development Steps

- Project setup: Empty Activity with appropriate naming
 - Resources: Update strings.xml with questions, answers, explanations
 - Styling: Amend themes.xml with custom styles
 - QuestionsFragment: Layout with questions and button interactions
 - AnswersFragment: Display results with detailed explanations
 - Optional: Multiple-choice fragment for answer selection

Chapter Summary

Comprehensive coverage of Android fragments from basic lifecycle understanding to advanced navigation patterns, establishing fragments as fundamental building blocks for modular Android development.

Key Concepts Mastered

- Fragment Lifecycle: Understanding creation, view setup, and destruction phases
- Static Fragments: XML-defined fragments with dual-pane layouts for responsive design
- Dynamic Fragments: Runtime fragment management with transactions and back stack
- Jetpack Navigation: Modern approach reducing boilerplate with navigation graphs
- Fragment Communication: Interface patterns for fragment-activity interaction

Practical Applications

- Code Reusability: Same fragments across different screen sizes and orientations
- Modular Development: Self-contained UI components with encapsulated logic
- Responsive Design: Automatic adaptation to phones, tablets, and different configurations
- Navigation Management: Structured user journeys with clear fragment relationships
- Modern Architecture: Foundation for advanced Android development patterns